Algebraic Domain Decomposition Methods for Darcy flow in heterogeneous media

Mikolaj Szydlarski

THESE DE DOCTORAT
DE L'UNIVERSITE PIERRE ET MARIE CURIE

ÉCOLE DOCTORALE DE SCIENCES MATHÉMATIQUES
DE PARIS CENTRE

Spécialité

MATHÉMATIQUES APPLIQUÉES

Présentée par

M. Mikołaj SZYDLARSKI

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse:

ALGEBRAIC DOMAIN DECOMPOSITION METHODS FOR DARCY FLOW IN HETEROGENEOUS MEDIA

Soutenue le 5 Novembre 2010 *devant le jury composé de*:

Président:	M. Frédéric HECHT	*Laboratoire J.L. Lions (UPMC)*
Rapporteur:	M. Bernard PHILIPPE	*INRIA (Rennes)*
Rapporteur:	M. Luc GIRAUD	*INRIA (Bordeaux)*
Directeur de thèse:	M. Frédéric NATAF	*Laboratoire J.L. Lions (UPMC)*
	M. Roland MASSON	*IFP Energies nouvelles*
	M. Pascal HAVÉ	*IFP Energies nouvelles*

“Receive my instruction, and not money: chose knowledge rather than gold. For wisdom is better than all the most precious things: and whatsoever may be desired cannot be compared to it.”

THE BOOK OF PROVERBS 8:10-11

To my father. The first true scientist I have met.

Acknowledgements

This thesis arose in part out of three years of research that has been done since I came to France. By that time, I have worked with a number of people whose contribution in assorted ways to the research and the making of the thesis deserved special mention. It is a pleasure to convey my gratitude to them all in my humble acknowledgment.

In the first place I would like to record my gratitude to Fédéric Nataf for his supervision, advice, and guidance as well as giving me extraordinary experiences through out the work. Above all and the most needed, he provided me unflinching encouragement and support in various ways. His scientist intuition and wisdom inspire and enrich my growth as a student, a researcher and a scientist want to be. I am indebted to him more than he knows.

Many thanks go in particular to Pascal Havé and Roland Masson. I am much indebted to Pascal for his valuable advice in science discussion, supervision in programming and furthermore, using his precious times to read this thesis and gave his critical comments about it. I have also benefited by supervision and guidance from Roland who kindly grants me his time for answering of my questions about the Black Oil model and porous media flow simulations.

I gratefully thank Bernard Philippe and Luc Giraud for their constructive comments on this thesis. I am thankful that in the midst of all their activity, they accepted to be members of the reading committee.

I would also acknowledge Thomas Guignon, whom I would like to thank for taught me how to work with IFP format for storage a sparse matrices originating from the real simulations of the porous media flow. It was a pleasure to work with an exceptionally experienced scientist like him.

Collective and individual acknowledgments are also owed to my colleagues at IFP whose present somehow perpetually refreshed, helpful, and memorable. Many thanks go in particular to Carole Widmer, Nataliya Metla, Eugenio Echague, Ivan Kapyrin, Gopalkrishnan Parameswaran, Riccardo Ceccarelli, Safwat Hamad, Hassan Fahs, Florian Haeberlein and Hoel Langouet for giving me such a pleasant time when working together with them since I knew them. Special thanks to Daniele Di Pietro for the coffee break meetings and dry humour about scientist's life. I also convey special acknowledgement to Sylvie Pegaz, Sylvie Wolf and Léo Agélas for being such good neighbours in the office, who always ready to lend me a hand. I would like also thank to Tao Zhao from LJLL for his generous help in performing a number of numerical experiments which results, thanks to his help some death-lines weren't so scary.

I was extraordinarily fortunate in having Chiara Simeoni as my master degree supervisor at the University of L'Aquila . I could never have embarked and started all of this without her prior teachings and wise advices. Thank you.

Where would I be without my family ? My parents deserve special mention for their inseparable support and prayers. My Father, Marcin Szydlarski, in the first place is the person who put the fundament my learning character, showing me the joy of intellectual pursuit ever since I was a child. My Mother, Danuta, is the one who sincerely raised me with her

caring and gently love. Karolina and Anna, thanks for being supportive and caring siblings.

Finally, I would like to thank everybody who was important to the successful realisation of thesis, as well as expressing my apology that I could not mention personally one by one.

Contents

Introduction

Definition of problem

Porous media flow simulations lead to the solution of complex non linear systems of coupled Partial Differential Equations (PDEs) accounting for the mass conservation of each component and the multiphase Darcy law. These PDEs are discretized using a cell-centered finite volume scheme and a fully implicit Euler integration in time in order to allow for large time steps. After Newton type linearization, one ends up with the solution of a linear system at each Newton iteration which all together represents up to 90 percents of the total simulation elapsed time. The linear systems couple an elliptic (or parabolic) unknown, the pressure, and hyperbolic (or degenerate parabolic) unknowns, the volume or molar fractions. They are non symmetric, and ill-conditioned in particular due to the elliptic part of the system, and the strong heterogeneities and anisotropy of the media. Their solution by an iterative Krylov method such as GMRES or BiCGStab requires the construction of an efficient preconditioner which should be scalable with respect to the heterogeneities, anisotropies of the media, the mesh size and the number of processors, and should cope with the coupling of the elliptic and hyperbolic unknowns.

In practice, a good preconditioner must satisfy many constraints. It must be inexpensive to compute and to apply in terms of both computational time and memory storage. Because we are interested in parallel applications, the construction and application of the preconditioner of the system should also be parallelizable and scalable. That is the preconditioned iterations should converge rapidly, and the performance should not be degraded when the number of processors increases. There are two classes of preconditioners, one is to design specialised algorithms that are close to optimal for a narrow type of problems, whereas the second is a general-purpose algebraic method. But this kind of preconditioning require a complete knowledge of the problem which may not always be feasible. Furthermore, these problem specific approaches are generally very sensitive to the details of the problem, and even small changes in the problem parameters can penalize the efficiency of the solver. On the other hand, the algebraic methods use only information contained in the coefficient of the matrices. These techniques achieve reasonable efficiency on a wide range of problems. In general, they are easy to apply and are well suited for irregular problems. Furthermore, one important aspect of such approaches is that they can be adapted and tuned to exploit

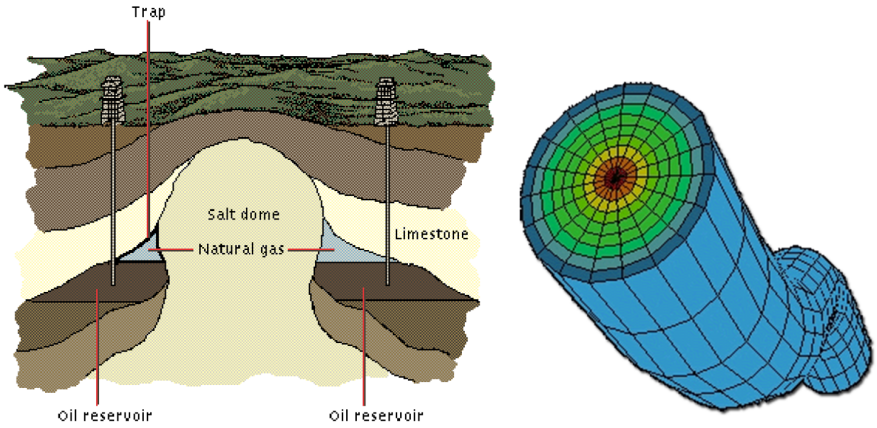


Figure 1: Example of oil reservoir and mesh of complex well

specific applications.

The current IFP solution is based on Algebraic MultiGrid preconditioners (AMG) for the pressure block combined with an incomplete ILU(0) factorisation of the full system. This so called combinative-AMG preconditioner is the most serious candidate for the new generation of reservoir simulators for its very good scalability properties with respect to the mesh size and the heterogeneities.

Nevertheless, this method still exhibits some problems of robustness when the linear system is too far from the algebraic multigrid paradigm (e.g. for strongly non diagonally dominant wells equations or for multipoint flux approximations or non linear closure laws leading to strongly positive off diagonal terms, or for convection dominated equations). Domain Decomposition Methods (DDM) are an alternative solution that could solve the above difficulties in terms of robustness and parallel efficiency on distributed architectures. These methods are naturally adapted to parallel computations and are more robust in particular when the subdomain problems are solved by a direct sparse solver. They also extend to coupled system of PDEs and enable to treat in the same framework the coupling of different type of models like wells equations or conductive faults.

Future of reservoir simulations

Continuously increasing demand for accurate results from reservoir simulations indicate usage of more dense and complex meshes along with advance numerical schemes which can deal with them. For example in current IFP models we have only one equation per well. However a new approach is developing in which a well has a complex modelling and refined mesh (see figure 1). In this context domain decomposition is not only an alternative but it becomes the natural choice for separating of model of flow around well and far from it.

Those future plans are another motivation for studying domain decomposition methods for multiphase, compositional porous media flow simulations.

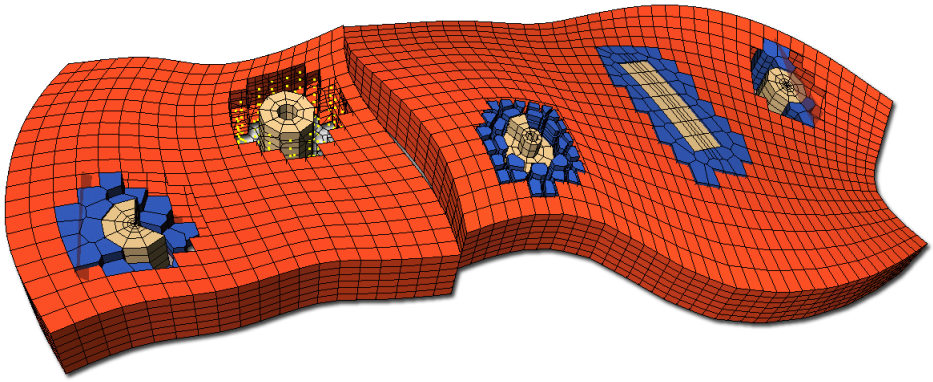


Figure 2: Example mesh of reservoir with complex wells (horizontal and vertical)

Objective

The objective of this PhD is to study and implement algebraic domain decomposition methods as a preconditioner of a Krylov iterative solver. This preconditioner could apply either on the full system or on the pressure "elliptic" block only as the second step of a combinative preconditioner. Therefore the main difficulties to be studied are the algebraic construction of interface conditions between the subdomains, the algebraic construction of a coarse grid, and the partitioning and load balancing within a distributed data structure implementation.

Context of work

This work is performed in the framework of *convention industrielle de formation par la recherche* (CIFRE) under the dual responsibility of *Pierre and Marie Curie University* (Paris VI) and *IFP Energies nouvelles* (IFP New Energy) [1]. Academic side is carried out by *Ecole Doctorale de Sciences Mathématiques de Paris Centre* on-site *Département Mathématiques Appliquées et le Laboratoire Jacques-Louis Lions*. From industrial point of view this thesis is a part of research project of IFP at department of Informatics and Applied Mathematics.

Plan of report

In chapter 1, we give an overview on the existing works on the two main ingredients of the domain decomposition methods: the interface conditions between the subdomains and the coarse grid corrections. As for the interface conditions, since the seminal paper by P.L. Lions [39], there has been many works on how to design efficient interface conditions. The problem can be considered at the continuous level and then discretized (see e.g. [19, 28, 44]). This

1. In June 2010 IFP (French Institute of Oil) changed name into IFP Energies nouvelles in order to more closely reflects IFP's objectives and the very nature of its research, with their increasing focus on new energy technologies.

approach, based on the use of the Fourier transform, is limited to smooth coefficients. In this chapter we focus on a method that works directly at the discrete level: the patch method [40]. As for the coarse grid correction, we explain that once the coarse space is chosen, the way to build the coarse grid correction is readily available from the papers by Nabben-Vuik and coauthors [57].

In chapter 2, we present the main features of the library that was developed in order to implement various existing methods and test new ideas. The library is carefully designed in C++ and MPI with a convenient parallel matrix storage that eases the test of new algorithms. Our data structure is very close to the one recently and independently proposed in [11]. Otherwise, the library uses as much as possible existing libraries (e.g. Metis and Scotch for partitioning, SuperLU, Hypre and PETSC for the sequential linear solvers). We report experiments made with this library testing existing domain decomposition methods: Schwarz methods with overlapping decompositions, partitioning with weights for taking into account anisotropy and discontinuities in an algebraic multigrid fashion, modified Schwarz method with interface conditions including the patch method [40]. The library will be used in the next chapters to test new methods in domain decomposition methods.

Chapter 3 introduces a new algebraic way to build interface conditions. It is shown in Lemma 3.1 that if the original matrix is symmetric positive definite (SPD), the local subproblems with the algebraic interface conditions will still be SPD. This construction depends on a parameter β (typically a diagonal matrix). In the two subdomain case, for a given harmonic vector in the subdomain, it is possible to build the interface condition such as to "kill" the error on this vector. This vector is chosen by computing Ritz eigenvectors from the Krylov space. In this respect, the method is adaptive and is very efficient for the two or three-subdomain case. But when there are many subdomains, numerical tests show that the method does not bring a benefit and it is thus limited to the two subdomain case.

This motivates chapter 4 where an adaptive coarse grid correction is introduced for the many subdomain case. Usually, the coarse space is given from an *a priori* analysis of the partial differential operator the equation comes from see [58] and references therein. For instance, in [47] when solving the Poisson equation it is suggested that the coarse space should consist of subdomain wise constant functions. For problems with discontinuous coefficients, this is usually not enough, see [46] and a richer coarse space is necessary. Another classical possibility in deflation methods is to make a first complete solve and to analyse then the Krylov space to build a meaningful coarse space for subsequent solves with the same matrix but a different right hand side. Here inspired by this method we propose a construction that is usable even before a first solve is completed. The principle is to compute Ritz eigenvectors responsible for a possible stagnation of the convergence. They are related to small eigenvalues of the preconditioned system. Then these global vectors are split domain-wise to build the coarse space Z. Its size is thus the number of Ritz eigenvectors times the number of subdomains. The coarse space is thus larger than the vector space spanned by the Ritz eigenvectors. It contains more information and can be used to complete more efficiently the first solve. Numerical results illustrate the efficiency of this approach even for problems with

discontinuous coefficients.

Finally in chapter 6, we give a conclusion.

Chapter 1

State of Art

In this chapter, formulation and origin of various Schwarz methods will be presented with emphasis on their algebraic formulation.

The widespread availability of parallel computers and their potential for the numerical solution of difficult to solve partial differential equations have led to large amount of research in domain decomposition methods. Domain decomposition methods are general flexible methods for the solution of linear or non-linear system of equations arising from the discretization of partial differential equations (PDEs). For the linear problems, domain decomposition methods can often be viewed as preconditioners for Krylov subspace techniques such as generalised minimum residual (GMRES). For non-linear problems, they may be viewed as preconditioners for the solution of the linear system arising from the use of Newton's method or as preconditioners for solvers. The term domain decomposition has slightly different meanings to specialists within the discipline of PDEs. In parallel computing it means the process of distributing data among the processors in a distributed memory computer. On the other hand in preconditioning methods, domain decomposition refers to the process of subdividing the solution of large linear system into smaller problems whose solutions can be used to produce a preconditioner (or solver) for the system of equations that results from the discretizing the PDE on the entire domain. In this context, domain decomposition refers only to the solution method for the algebraic system of equations arising from discretization. Finally in some situations, the domain decomposition is natural from the physics of the problem: different physics in different subdomains, moving domains or strongly heterogeneous media. Those separated regions can be modelled with different equations, with the interfaces between the domains handled by various conditions. Note that all three of these may occur in a single program. We can conclude that the most important motivations for a domain decomposition method are their ease of parallelization and good parallel performance as well as simplification of problems on complicated geometry.

Many domain decomposition algorithms have been developed in the past few years, however there is still a lack of black-box routines working at the matrix level which could

lead to the widespread adoption of these techniques in engineering and scientific computing community. One of the goals of this thesis is to follow a path which leads to construction of such black-box solver by a collaboration between numerical analysis and computer science.

1.1 Original Schwarz Methods

The earliest known domain decomposition method was invented by Hermann Amandus Schwarz dating back to 1869 [53]. He studied the case of a complex domain decomposed into two subdomains, which are geometrically much simpler, namely a disc Ω_1 and rectangle Ω_2, with interfaces $\Gamma_1 := \partial\Omega_1 \cap \Omega_2$ and $\Gamma_2 := \partial\Omega_2 \cap \Omega_1$, on which he wished to solve:

$$\begin{aligned} -\Delta(u) &= f \quad \text{in} \quad \Omega \\ u &= g \quad \text{on} \quad \partial\Omega. \end{aligned} \tag{1.1}$$

Schwarz proposed an iterative method (called now *alternating Schwarz method*) which only uses solution on the disk and the rectangle. The method starts with an initial guess u_1^0 along Γ_1 and then computes iteratively for $n = 0, 1, \ldots$ the iterates u_1^{n+1} and u_2^{n+1} according to the algorithm

Alternating Schwarz Method

$$\begin{aligned} -\Delta(u_1^{n+1}) &= f \quad \text{in} \quad \Omega_1 \\ u_1^{n+1} &= g \quad \text{on} \quad \partial\Omega_1 \backslash \Gamma_1 \\ u_1^{n+1} &= u_2^{n} \quad \text{on} \quad \Gamma_1. \end{aligned} \qquad \begin{aligned} -\Delta(u_2^{n+1}) &= f \quad \text{in} \quad \Omega_2 \\ u_2^{n+1} &= g \quad \text{on} \quad \partial\Omega_2 \backslash \Gamma_2 \\ u_2^{n+1} &= u_1^{n+1} \quad \text{on} \quad \Gamma_2. \end{aligned} \tag{1.2}$$

Schwarz proved that the sequence of functions u_1^n and u_2^n converge uniformly and they agree on both Γ_1 and Γ_2, and thus they must be identical in the overlap. He therefore concludes that u_1 and u_2 must be values of the same function u which satisfy (1.1) on Ω.

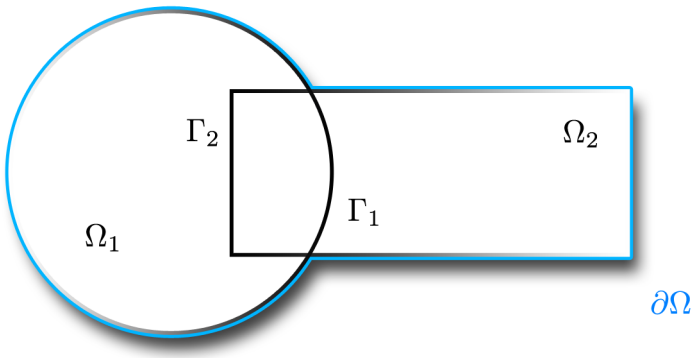


Figure 1.1: An example of two overlapping subdomains with two artificial interfaces.

This algorithm was carefully studied by Pierre Louis Lions in [39] where he also proved convergence of the "parallel" version of the original Schwarz algorithm [39]:

> "The final extension we wish to consider, concerns the "parallel" version of the Schwarz alternating method ..., u_i^{n+1} is solution of $-\Delta u_i^{n+1} = f$ in Ω_i and $u_i^{n+1} = u_j^n$ on $\partial\Omega_i \cap \Omega_j$."

In contrast to *alternating Schwarz method* we call this method the *parallel Schwarz method* which is given by:

Parallel Schwarz Method

$$\begin{array}{rclcl|rclcl} -\Delta(u_1^{n+1}) & = & f & \text{in} & \Omega_1 & -\Delta(u_2^{n+1}) & = & f & \text{in} & \Omega_2 \\ u_1^{n+1} & = & g & \text{on} & \partial\Omega_1\backslash\Gamma_1 & u_2^{n+1} & = & g & \text{on} & \partial\Omega_2\backslash\Gamma_2 \\ u_1^{n+1} & = & u_2^n & \text{on} & \Gamma_1. & u_2^{n+1} & = & u_1^n & \text{on} & \Gamma_2. \end{array} \tag{1.3}$$

The only change is the iteration index in the second transmission condition. For given initial guesses u_1^0 and u_2^0, problems in domains Ω_1 and Ω_2 for $n = 0, 1, \ldots$ may be solved concurrently and so the new algorithm (1.3) is parallel and thus well adapted to parallel computers.

1.1.1 Discrete Schwarz Methods

Writing the system (1.1) for the discretized problem by a finite difference, finite volume or finite elements methods, yields a linear system of the form

$$\mathrm{AU} = \mathrm{F}. \tag{1.4}$$

Where F is a given righthand side, U is the set of unknowns and A is the discretization matrix. Schwarz methods have also been introduced directly at the algebraic level for such linear systems, and there are several variants.

In order to obtain a domain decomposition for (1.4), one needs to decompose the unknowns in the vector U into subsets corresponding to subdomains on continuous level. To quantify this operation, we need to introduce some notation. For the sake of simplicity, we consider only a two subdomain case. But, the ideas carry over easily to the general case. Let $\mathcal{N}_i$, $i = 1, 2$ be a partition of the indices corresponding to the vector U. Let R_i, $i = 1, 2$ denote the matrix that when applied to the vector U returns only those values associated with the nodes in $\mathcal{N}_i$. When we consider for instance a domain 1 made of the nodes 1,2 and 4, the matrix R_1 is given by

$$\mathrm{R}_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & 0 & \cdots & 0 \end{pmatrix} \tag{1.5}$$

The transpose of R_1 simply inserts the given values u into the larger array

$$\begin{pmatrix} v_1 & v_2 & 0 & v_3 & 0 & \cdots & 0 \end{pmatrix}^{\mathrm{T}} = \mathrm{R}_1^{\mathrm{T}} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}$$

The matrices R_i $i = 1,2$ are often referred to as the *restriction* operators, while R_i^T are the *interpolation* matrixes. With these restriction matrices, $R_j U = U_j$ $(j = 1,2)$ gives decomposition set of unknowns for our two domain case. One can also define restriction on the matrix A to the first and second unknowns using the same restriction matrices,

$$A_j = R_j A R_j^T, \quad j = 1,2. \tag{1.6}$$

Thus the matrix $R_j A R_j^T$ is simply the subblock of A associated with the given nodes.

Using this form we can write the *multiplicative Schwarz method* (MSM) in two fractional steps:

$$\begin{aligned} U^{n+\frac{1}{2}} &= U^n + R_1^T A_1^{-1} R_1 (F - AU^n) \\ U^{n+1} &= U^{n+\frac{1}{2}} + R_2^T A_2^{-1} R_2 (F - AU^{n+\frac{1}{2}}). \end{aligned} \tag{1.7}$$

Since each iteration involves sequential fractional steps, this is not ideal solution for parallel computing, contrary to the *Additive Schwarz Method* (ASM) algorithm, introduced by Dryja and Widlund [20]:

> "The basic idea behind the additive form of the algorithm is to work with the simplest possible polynomial in the projections. Therefore the equation $(P_1 + P_2 + \ldots + P_N) u_h = g'_h$ is solved by an iterative method."

Thus using the same notation as for MSM in our two-subdomain model problem, the preconditioned system proposed by Dryja and Widlund is as follow:

Additive Schwarz Method as a preconditioner

$$\left(R_1^T A_1^{-1} R_1 + R_2^T A_2^{-1} R_2\right) AU = \left(R_1^T A_1^{-1} R_1 + R_2^T A_2^{-1} R_2\right) F. \tag{1.8}$$

Using this preconditioner for a stationary iterative method yields

$$U^{n+1} = U^n + \left(R_1^T A_1^{-1} R_1 + R_2^T A_2^{-1} R_2\right) (F - AU^n). \tag{1.9}$$

We can extend this idea immediately to methods that involve more than two subdomains. For a domain $\Omega = \cup \Omega_j$, (1.9) can be written as

$$U^{n+1} = U^n + \sum_j \widetilde{B}_j (F - AU^n) \quad \text{with} \quad \widetilde{B}_j = R_j^T A_{\Omega_j}^{-1} R_j \quad \text{and} \quad A_{\Omega_i} = R_j A R_j^T. \tag{1.10}$$

A (1.9) algorithm resembles the *parallel Schwarz method* (1.3) but it is not equivalent to a discretization of Lions's parallel method, except if R_j are non-overlapping in algebraic sense. Thus if $R_1^T R_1 + R_2^T R_2 \neq \mathbb{1}$ method can fail to converge, as it has been showed for Poisson equation in [22]. However with some special treatment like a relaxation parameter [43] method still converge but this so called "damping factor" and its size is strongly connected with problem of the method in the overlap (see [26] for instance).

Nevertheless, the preconditioned system (1.8) has very desirable properties for solution with a Krylov method: the preconditioner is symmetric, if A is symmetric. Including a coarse

grid correction denoted by $ZE^{-1}Z^T$ (see §1.4 for definition) in the additive Schwarz preconditioner, we obtain

$$\mathscr{P}_{as} := \sum_{j=1}^{J} R_j^T A_{\Omega_j}^{-1} R_j + ZE^{-1}Z^T. \tag{1.11}$$

Dryja and Widlund showed in [20] a fundamental condition number estimate for this preconditioner applied to the Poisson equation, discretized with characteristic coarse mesh H, fine mesh size h and an overlap δ:

Theorem 1.1. *The condition number* κ *of operator* A*, preconditioned by* $\mathscr{P}_{as}$ *i.e., ASM* (1.8) *with the coarse grid correction, satisfies*

$$\kappa(\mathscr{P}_{as}A) \leq C\left(1 + \frac{H}{\delta}\right), \tag{1.12}$$

where the constant C *is independent of* h*,* H *and* δ*.*

Thus the additive Schwarz method used as a preconditioner for a Krylov method seems to be optimal in sense that it converges independently of the mesh size and the number of subdomains, if the ratio of H and δ is constant. However in 1998, a new family of Schwarz methods was introduced by chance by Cai and Sarkis [12]:

> "While working on an AS/GMRES algorithm in an Euler simulation, we removed part of the communication routine and surprisingly the "then AS" method converged faster in both terms of iteration counts and CPU time."

When we use the same notation as before for our two-subdomain case, the *restricted additive Schwarz* (RAS) iterations is

$$U^{n+1} = U^n + \left(\widetilde{R}_1^T A_1^{-1} R_1 + \widetilde{R}_2^T A_2^{-1} R_2\right)(F - AU^n) \tag{1.13}$$

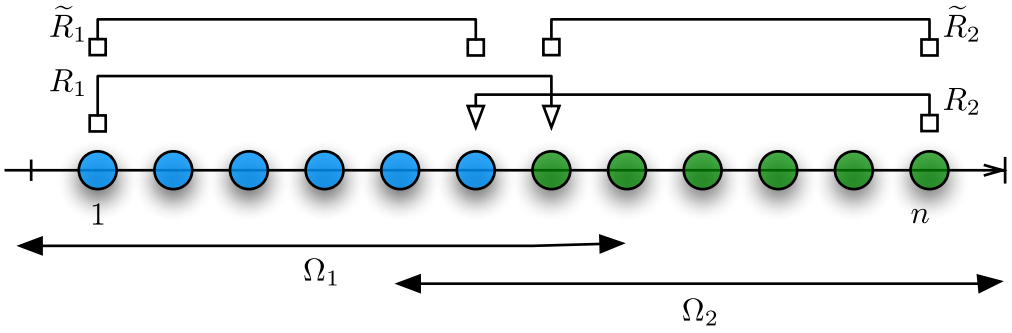
where new restriction matrices $\widetilde{R}_j$ correspond to non-overlapping decomposition, so that $\widetilde{R}_1^T\widetilde{R}_1 + \widetilde{R}_2^T\widetilde{R}_2 = \mathbb{1}$, the identity. For an illustration in one and two dimensions, see Figure 1.2.

As in the case of additive Schwarz method we can extend this idea to methods that involve J subdomains,

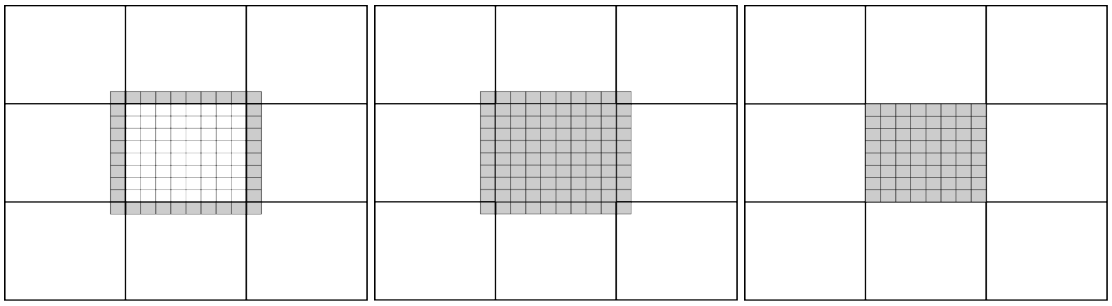
$$U^{n+1} = U^n + \sum_{j=1}^{J} \widetilde{R}_j^T A_{\Omega_j}^{-1} R_j (F - AU^n). \tag{1.14}$$

This is proved in [26] that the RAS method is equivalent to a discretization of parallel Schwarz method (1.3). However there is no convergence theorem similar to Theorem 1.1 for restricted additive Schwarz. There are only comparison results at the algebraic level between additive and restricted Schwarz [22]:

> "Using a continuos interpretation of the RAS preconditioner we have shown why RAS has better convergence properties than AS. It is due to the fact that, when used as iterative solvers, RAS is convergent everywhere, whereas AS is not convergent in the overlap. Away from the overlap, the iterates are identical. This observation holds not only for discretized partial differential equations, it is true for arbitrary matrix problem."



(a) The restrictions operators for a one dimensional example.



(b) Subdomain with overlap (c) The restriction operator R_i (d) The restriction operator $\widetilde{R_i}$

Figure 1.2: Graphical representation of the restriction operators in RAS

Unfortunately, the restricted additive Schwarz preconditioner is non-symmetric, even if the underlying system matrix A is symmetric, and hence a Krylov method for non-symmetric problems needs to be used. For more details follow Cai and Sarkis in [12] or Efstathiou and Gander in [22] and bibliography therein.

1.1.2 Drawbacks of original Schwarz methods

As we can see Schwarz algorithms brings some benefits to parallel computation techniques. For example their fundamental idea of decomposition the original problem into smaller pieces reduce amount of storage but if we take into account CPU usage, the original algorithms are very slow (e.g. in comparison with multi-grid method [9]). The other big drawback of the classical Schwarz method is in their need of overlap in order to converge. This is not only a drawback in sense that we waste efforts in the region shared by the subdomains but for example in problems with discontinuous coefficients, a non-overlapping decomposition with the interface along discontinuity would be more natural.

Lions therefore proposed a modification of the alternating Schwarz method for a non-overlapping decomposition, as illustrated in Figure 1.3. The Dirichlet interface conditions on Γ ($\partial\Omega_i \backslash \partial\Omega$, $i = 1,2$) have been replaced by *Robin interface condition* ($\partial\Omega n_i + \alpha$, where n is the outward normal to subdomains Ω_i). With this new Robin transmission condition, Lions proved in [39] that the new Schwarz method is convergent without overlap for the case of

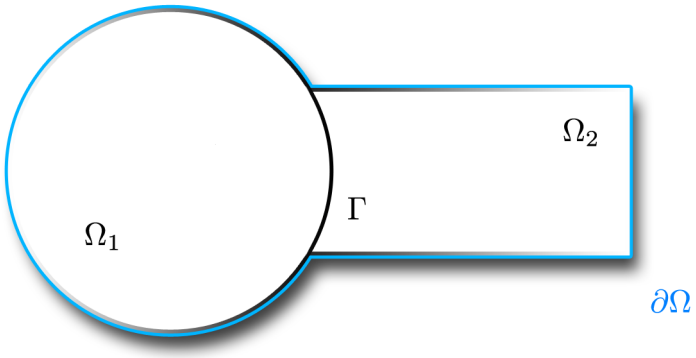


Figure 1.3: An example of two non-overlapping subdomains with artificial interface.

constant parameter α and an arbitrary number of subdomains. However from his analysis one can not see how the performance depends on the parameters α, but Lions showed that for one dimensional model problem, one can choose the parameters in such way that the method with two subdomains converges in two iterations, which transforms this iterative method into a direct solver.

Moreover Lions (and independently Hagstrom, Tewarson and Jazcilevich [33]) stated in [39] that even more general interface conditions can be defined on the interface:

> "First of all, it is possible to replace the constants in the Robin condition by two proportional functions on the interface, or even by local or nonlocal operators."

His seminal paper has been the basis for many other works [2, 3, 38] which showed that for all the drawbacks of the classical Schwarz methods, significant improvements have been achieved by modifying the transmission conditions. That has led to a new class of Schwarz methods which we call now *optimized Schwarz methods*.

1.2 Optimal Interface Condition

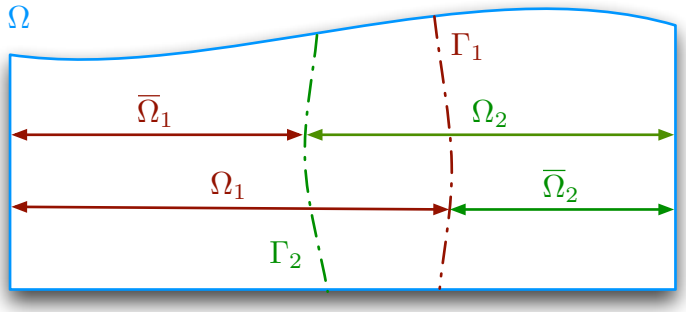
As it has been mentioned in the previous section, the major improvements of Schwarz methods come from the use of the other interface condition. The convergence proof given by P. L. Lions in the elliptic case was extended by B. Després to the Helmholtz equation in [19] (general presentation can be found in [17]). A general convergence for interface condition with second order tangential derivatives has been proved. However it gives the general condition in an *a priori* form. From numerical point of view it would be more practical to derive them so as they have the fastest convergence. It was done by F. Nataf, F. Rogier and E. de Sturler [45]:

> "The rate of convergence of Schwarz and Schur type algorithms is very sensitive to the choice of interface condition. The original Schwarz method is based

> on the use of Dirichlet boundary conditions. In order to increase the efficiency of the algorithm, it has been proposed to replace the Dirichlet boundary condition with more general boundary conditions. ...It has been remarked that absorbing (or artificial) boundary conditions are a good choice. In this report, we try to clarify the question of the interface condition."

They consider a general linear second order elliptic partial operator $\mathscr{L}$ and regular, arbitrary in number of subdomains, decomposition of domain Ω. For the sake of simplicity we present this result for two domain case.

Problem 1.1. *Find u such that $\mathscr{L}(u) = f$ in a domain Ω and $u = 0$ on $\partial\Omega$. The domain Ω is decomposed into two subdomains Ω_1 and Ω_2. We suppose that the problem is regular so that $u_i := u|_{\Omega_i}$, $i = 1,2$, is continuous and has continuous normal derivatives across the interface $\Gamma_i = \partial\Omega_i \cap \overline{\Omega}_j$, $i \neq j$.*



A modified, by new interface condition, Schwarz type method is considered now as:

$$\begin{aligned}
\mathscr{L}(u_1^{n+1}) &= f && \text{in} \quad \Omega_1 \\
u_1^{n+1} &= 0 && \text{on} \quad \partial\Omega_1 \cap \partial\Omega \\
\mu_1 \nabla u_1^{n+1} \cdot \mathbf{n_1} &+ \mathscr{B}_1(u_1^{n+1}) && \\
= -\mu_1 \nabla u_2^{n} \cdot \mathbf{n_2} &+ \mathscr{B}_1(u_2^{n}) && \text{on} \quad \Gamma_1 \\
& && \\
\mathscr{L}(u_2^{n+1}) &= f && \text{in} \quad \Omega_2 \\
u_2^{n+1} &= 0 && \text{on} \quad \partial\Omega_2 \cap \partial\Omega \\
\mu_2 \nabla u_2^{n+1} \cdot \mathbf{n_2} &+ \mathscr{B}_2(u_2^{n+1}) && \\
= -\mu_2 \nabla u_1^{n} \cdot \mathbf{n_1} &+ \mathscr{B}_2(u_1^{n}) && \text{on} \quad \Gamma_2
\end{aligned} \tag{1.15}$$

where μ_1 and μ_2 are real-valued functions and $\mathscr{B}_1$ and $\mathscr{B}_2$ are operators acting along the interfaces Γ_1 and Γ_2. For instance, $\mu_1 = \mu_2 = 0$ and $\mathscr{B}_1 = \mathscr{B}_2 = \mathbb{1}$ correspond to the parallel Schwarz algorithm (1.3); $\mu_1 = \mu_2 = 1$ and $\mathscr{B}_i = \alpha \in \mathbb{R}$, $i = 1,2$, has been proposed in [39] by P. L. Lions.

The authors proved that use of non-local DtN (Dirichlet to Neumann) map (a.k.a. Steklov-Poincaré) as interface conditions in (1.15) ($\mathscr{B}_i = \mathrm{DtN}_j$ $(i \neq j)$) is optimal and leads to (*exact*)

convergence in two iterations. The main feature of this result is to be very general since it does not depend on the exact form of the operator $\mathscr{L}$ and can be extended to system or to coupled systems of equations, despite that they are not practical because of its non-local nature, thus the new algorithm (1.15) is much more costly to run and difficult to implement. Nevertheless, this result is a guide for a choice of partial interface conditions (e.g. as a "object" to aproximate). Moreover, this result establish a link between the optimal interface condition and artificial boundary conditions.

Definition 1.1 (DtN map)**.** *Let*

$$\begin{aligned} u_0 &: \Gamma_1 \to \mathbb{R} \\ \mathrm{DtN}_2(u_0) &:= \nabla v \cdot n_{2_{|\partial\Omega_1 \cap \overline{\Omega}_2}}, \end{aligned} \tag{1.16}$$

where n_2 is the outward normal to $\Omega_2 \setminus \overline{\Omega}_1$, and v satisfies the following boundary value problem:

$$\begin{array}{rcl} \mathscr{L}(v) & = & 0 \quad in \quad \Omega_2 \setminus \overline{\Omega}_1 \\ v & = & 0 \quad on \quad \partial\Omega_2 \cap \partial\Omega \\ v & = & u_0 \quad on \quad \partial\Omega_1 \cap \overline{\Omega}_2. \end{array}$$

1.3 Optimised Schwarz Method

Optimised Schwarz method is obtained from the classical one by changing the transmission condition. In the discrete Schwarz method however, the transmission condition do not appear naturally anymore. One can however show algebraically that is suffices to replace the subdomains matrices A_j in additive or restricted Schwarz method by subdomain matrices representing discretization of subdomain problems with Robin or more general (e.g. optimal) boundary conditions.

1.3.1 Optimal Algebraic Interface Conditions

When the problem (1.1) is discretized by a finite element or a finite difference method, it yields a linear system $\mathrm{AU} = \mathrm{F}$. If domain Ω in our problem is decomposed into two subdomains Ω_1 and Ω_2, at the discrete level this decomposition leads to the matrix partitioning

$$\begin{pmatrix} \mathrm{A}_{11} & \mathrm{A}_{1\Gamma} & 0 \\ \mathrm{A}_{\Gamma 1} & \mathrm{A}_{\Gamma\Gamma} & \mathrm{A}_{\Gamma 2} \\ 0 & \mathrm{A}_{2\Gamma} & \mathrm{A}_{22} \end{pmatrix} \begin{pmatrix} \mathrm{U}_1 \\ \mathrm{U}_\Gamma \\ \mathrm{U}_2 \end{pmatrix} = \begin{pmatrix} \mathrm{F}_1 \\ \mathrm{F}_\Gamma \\ \mathrm{F}_2 \end{pmatrix}. \tag{1.17}$$

where U_Γ corresponds to the unknowns on the interface Γ, and $\mathrm{U}_j, j = 1,2$ represent the unknowns in the interior of subdomains Ω_1 and Ω_2. In order to write a "modified" (by new interface condition) Schwarz method, we have to introduce two square matrixes S_1 and S_2 which act on vectors of the type U_Γ, then the modified Schwarz method reads:

$$\begin{pmatrix} \mathrm{A}_{11} & \mathrm{A}_{1\Gamma} \\ \mathrm{A}_{\Gamma 1} & \mathrm{A}_{\Gamma\Gamma} + \mathrm{S}_2 \end{pmatrix} \begin{pmatrix} \mathrm{U}_1^{n+1} \\ \mathrm{U}_{\Gamma,1}^{n+1} \end{pmatrix} = \begin{pmatrix} \mathrm{F}_1 \\ \mathrm{F}_\Gamma + \mathrm{S}_2 \mathrm{U}_{\Gamma,2}^n - \mathrm{A}_{\Gamma 2} \mathrm{U}_2^n \end{pmatrix} \tag{1.18a}$$

$$\begin{pmatrix} A_{22} & A_{2\Gamma} \\ A_{\Gamma 2} & A_{\Gamma\Gamma}+S_1 \end{pmatrix}\begin{pmatrix} U_2^{n+1} \\ U_{\Gamma,2}^{n+1} \end{pmatrix} = \begin{pmatrix} F_2 \\ F_\Gamma + S_1 U_{\Gamma,1}^n - A_{\Gamma 1} U_1^n \end{pmatrix} \tag{1.18b}$$

Lemma 1.1. *Assume* $A_{\Gamma\Gamma}+S_1+S_2$ *is invertible and problem* (1.17) *is well-posed. Then if the algorithm* (1.18) *converges, it converges to the solution of* (1.17)*. This is to be understood in the sense that if we denote by* $(U_1^\infty, U_{\Gamma,1}^\infty, U_2^\infty, U_{\Gamma,2}^\infty)$ *the limit as n goes to infinity of the sequence* $(U_1^n, U_{\Gamma,1}^n, U_2^n, U_{\Gamma,2}^n)_{n\geqslant 2}$ *we have for* $i = 1,2$ *:*

$$U_i^\infty = U_i \quad \textit{and} \quad U_{\Gamma,1}^\infty = U_{\Gamma,2}^\infty = U_\Gamma$$

Remark 1.1. *Note that we have a duplication of the interface unknowns* U_Γ *into* $U_{\Gamma,1}$ *and* $U_{\Gamma,2}$*.*

Proof. We subtract the last line of (1.18a) to the last line of (1.18b) as n goes to infinity shows that $(U_1^\infty, U_{\Gamma,1}^\infty = U_2^\infty, U_{\Gamma,2}^\infty)^T$ is a solution to (1.17) which is unique by assumption. □

Now following F. Magoulès, F.-X. Roux and S. Salmon in [40] we define optimal interface condition (as the discrete counterparts of DtN maps)

Lemma 1.2. *Assume* A_{ii} *is invertible for* $i = 1,2$*. Then in algorithm (1.18a)-(1.18b), taking*

$$S_1 = -A_{\Gamma 1} A_{11}^{-1} A_{1\Gamma}$$

and

$$S_2 = -A_{\Gamma 1} A_{22}^{-1} A_{2\Gamma}$$

yields a convergence in two steps.

Proof. Notice that in this case, the bottom-right blocks of the two by two block matrices in (1.18a) and (1.18b) are Schur complements. It is classical that the subproblems (1.18a) and (1.18b) are well-posed.

By linearity, in order to prove convergence, it is sufficient to consider the convergence to zero in the case $(F_1, F_{\Gamma,1}, F_2)^T = 0$. At step 1 of the algorithm, we have

$$A_{11} U_1^1 + A_{1\Gamma} U_{\Gamma,1}^1 = 0$$

or equivalently by applying $-A_{\Gamma 1} A_{11}^{-1}$:

$$-A_{\Gamma 1} U_1^1 - A_{\Gamma 1} A_{11}^{-1} A_{1\Gamma} U_{\Gamma,1}^1 = 0$$

So that the righthand side of (1.18b) is zero at step 2 (i.e. $n = 1$). We have thus convergence to zero in domain 2. The same proof holds for domain 1. □

Unfortunately the matrices S_1 and S_2 are in general dense matrixes whose computation and use as interface conditions is very costly.

1.3.2 Patch Method

In the previous sections it has been recalled that the best choice for absorbing boundary conditions is linked with Dirichlet-to-Neumann (DtN) maps of the outside of each subdomain on its interface boundary. After discretization, this choice leads to a dense augmented matrix equal to the complete outer Schur complement matrix (see §1.3). Some approximation techniques are required to obtain better performance in terms of computing time and CPU usage.

Several algebraic techniques of approximation of DtN map have been proposed by F. Magoulès, F-X.Roux and L. Series in [41]. They are based on the computation of a small and local DtN maps in order to approximate the complete and non-local operator DtN for the equation of linear elasticity. In elasticity DtM map i.e., the outer Schur complement matrix models the stiffness of all the outer sub-domain, hence the "first step" in its approximation leads to the restriction of the information only to the neighbouring sub-domains. This implies to consider the Schur complement matrix of the neighbouring subdomains only.

Unfortunately the neighbour Schur complement matrix is still a dense matrix hence using it as an augmented matrix increases the bandwidth (this implies a lot of additional operations during the factorisation of subdomain matrix). To reduce this cost, a simple sparse mask of the neighbour Schur complement matrix can be considered. This technique presents the advantage of keeping the sparsity of the subdomain matrix after addition of the augmented sub-operator. In order to construct this sparse mask we can form small successive parts of nodes (the patch) of subdomain as an entire system. For this purpose, new subsets of indices are defined for the nodes of the subdomain:

$$
\begin{array}{lcl}
\upsilon & = & \textit{indices of nodes inside the subdomain and on the interface,} \\
\upsilon_\Gamma & = & \textit{indices of nodes on the interface } \Gamma, \\
\upsilon_p^i & = & \textit{indices of nodes belonging to } \upsilon_\Gamma \textit{ such that the minimum connectivity} \\
 & & \textit{distance between each of these nodes and the node labelled } i \textit{ is lower} \\
 & & \textit{then } p, p \in \mathbb{N}, \\
\upsilon_{p,l}^i & = & \textit{indices of nodes belonging to } \upsilon \textit{ such that the minimum connectivity} \\
 & & \textit{distance between each of these nodes and the nodes belonging to } \upsilon_p^i \textit{ is} \\
 & & \textit{lower then } l,\ l \in \mathbb{N}.
\end{array}
$$

The subset υ_p^i corresponds to a patch of radius p around the node labeled i. Thus the subset $\upsilon_{p,l}^i$ corresponds to a neighboring area of width p and depth l of the patch. The sparse approximation consists of defining a sparse augmented matrix

$$
\mathrm{A}_{\mathrm{P}}^j = \begin{pmatrix} \overline{\mathrm{A}}_{jj} & \overline{\mathrm{A}}_{j\Gamma} \\ \overline{\mathrm{A}}_{\Gamma j} & \overline{\mathrm{A}}_{\Gamma\Gamma}^j \end{pmatrix} \tag{1.19}
$$

which obeys the same laws as the original problem because is defined by extracting rows and columns from general matrix. We can create such systems for each node on interface, in order to use them for computation "small" DtN maps from which we next, extract the coefficients of the line associated with interface node and we insert them inside the matrix

Algorithm 1 Sparse approximation of neighbour Schur complement

Require: Initialise p and l

Ensure: $(p \geq 1)$ and $(l \geq 1)$

1: Construct sparse matrix structure of interface matrix $\overline{\mathrm{S}}^{j}_{\Gamma\Gamma} \in \mathbb{R}^{dim(\upsilon_\Gamma)\times dim(\upsilon_\Gamma)}$,
2: Construct sparse matrix structure of sub-domain matrix $\mathrm{A}^{j} \in \mathbb{R}^{dim(\upsilon)\times \dim(\upsilon)}$,
3: Assembly the matrix A^{j}
4: **for all** $i \in \upsilon_\Gamma$ **do**
5: Extract coefficients A^{j}_{mn}, $(m,n) \in \upsilon^{i}_{p,l} \times \upsilon^{i}_{p,l}$, and construct the sparse matrix $\mathrm{A}^{j}_{\mathrm{P}} \in \mathbb{R}^{dim(\upsilon^{i}_{p,l}\times\upsilon^{i}_{p,l})}$ with these coefficients.
6: Compute the matrix $\widetilde{\mathrm{S}}^{j}_{i} = -\overline{\mathrm{A}}_{\Gamma j}(\overline{\mathrm{A}}_{jj})^{-1}\overline{\mathrm{A}}_{j\Gamma}$.
7: Extract the coefficients of the line associated with the node i from the matrix $\widetilde{\mathrm{S}}^{j}_{i}$ and insert them inside the matrix $\overline{\mathrm{S}}^{j}_{\Gamma\Gamma}$ at the line associated with the node i.
8: **end for**
9: Construct the symmetric matrix $\overline{\mathrm{S}}^{j}_{\Gamma\Gamma} = \frac{1}{2}\left(\overline{\mathrm{S}}^{j\ \mathrm{T}}_{\Gamma\Gamma} + \overline{\mathrm{S}}^{j}_{\Gamma\Gamma}\right)$

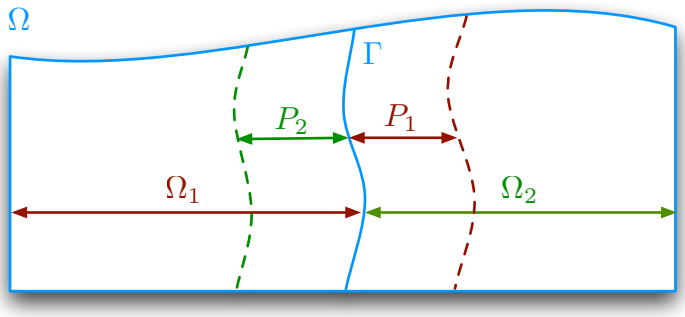
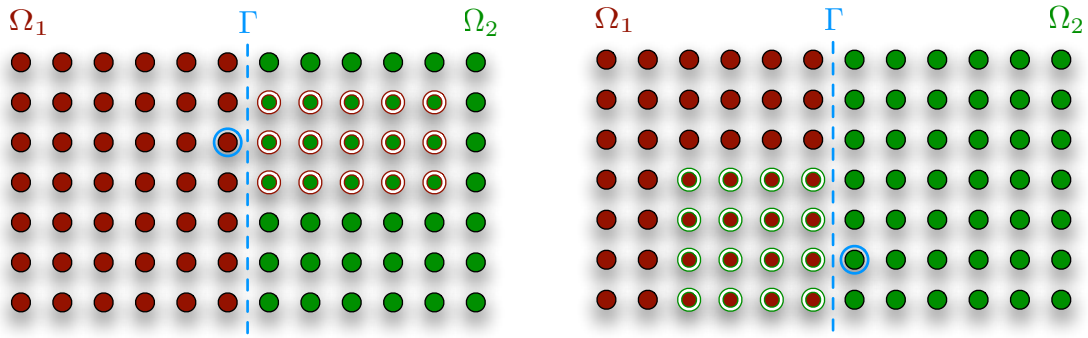


Figure 1.4: Non-overlaping domain decomposition, with patches P_1 and P_2.



(a) The example of patch P_1 with width $p = 1$ and deep $l = 5$

(b) The example of patch P_2 with width $p = 2$ and deep $l = 4$

Figure 1.5: An example of subdomain with 2D patches.

$\overline{S}^j_{\Gamma\Gamma}$. Thus a matrix $\overline{S}^j_{\Gamma\Gamma}$ can be used instead of S_j in algorithm (1.18). The complete procedure is presented in Algorithm (1).

The analysis of those methods by M. J. Gander, L. Halpern, F. Magoulès and F-X. Roux in [27] shows that this particular case of the geometric patch method, namely, the case of one patch per subdomain interface with width $p \simeq \infty$, leads to an algorithm equivalent to overlaping Schwarz method with Dirichlet to Neumann transmission condition at the new interface location defined by the end of the patches. Algebraic patch methods can be constructed without geometric information from underlying mesh, directly based on the matrix, and their convergence depends on the size of the patches, which represents the overlap of the equivalent classical Schwarz method. Hence algebraic patch substructing methods converge independently of the mesh parameter if the patch size is constant in physical space, which has been proved by numerical tests.

1.4 Two-level domain decomposition method

Single level methods are effective only for a small number of subdomains. The problem with single level methods is that the information about f and g in (1.1) in one subdomain traverse through all the intermediate subdomains only through their common interfaces. Thus for example the convergence rate of the single level adaptive Schwarz method becomes progressively worse when the number of subdomains becomes large. From an algebraic point of view this loss of efficiency is caused by the presence of small eigenvalues in the spectrum of the preconditioned, coefficient matrix. They have a harmful influence on the condition number, thus in addition to traditional preconditioner like ASM (1.8), a second kind of preconditioner can be incorporated to improve the conditioning of the coefficient matrix, so that the resulting approach gets rid of the effect of both small and large eigenvalues. This combined preconditioning is also know as 'two-level preconditioning', and the resulting iterative method is called a 'projection method' [57].

Simple example of two-level preconditioned system is Conjugate Gradient method (CG) [35, 51] combined with a two-grid method. In this case, together with the fine-grid linear system from which the approximate solution of the original differential equations is computed, a coarse-grid system is build based on a predefined coarse grid. From a Multi Grid (MG) [9] point of view, the (second-level) coarse-grid system is used to reduce the slow-varying, low frequency components of the error, that could not be effectively reduce on the (first-level) fine grid. These low frequency components of the error are associated with the small eigenvalues of the coefficient matrix. The high frequency components are, however, effectively handled on the fine grid. The latter is associated with the large eigenvalues of the coefficient matrix.

In order to define projection method we need to introduce some terminology:

Definition 1.2. *Suppose that an SPD coefficient matrix,* $A \in \mathbb{R}^{n\times n}$*, a right hand side,* $F \in \mathbb{R}^n$*, and SPD preconditioning matrix,* $M^{-1} \in \mathbb{R}^{n\times n}$*, and deflation subspace matrix,* $Z \in \mathbb{R}^{n\times k}$*, with full rank and* $k \leq n$ *are given. Then, we define the invertible matrix* $E \in \mathbb{R}^{k\times k}$*, the matrix* $Q \in \mathbb{R}^{n\times n}$*, and the deflation matrix,* $P \in \mathbb{R}^{n\times n}$*, as follows:*

$$P := \mathbb{1} - AQ \quad Q := ZE^{-1}Z^T \quad E := Z^TAZ,$$

where $\mathbb{1}$ *is the* $n \times n$ *identity matrix.*

Note that E is SPD for any full-rank Z, since A is SPD. Moreover, if $k \ll n$, then E is a matrix with small dimension, thus it can be easily computed and factored.

All operators of the projection methods consist of an arbitrary preconditioner M^{-1}, combined with one or more matrices P and Q. Thus in projection methods used in DDM, preconditioner M^{-1} consist of the local exact or inexact solvers on subdomains. Thereby we define it as in (1.8). The matrix Z describes a restriction operator, while Z^T is prolongation operator based on the subdomains and we define it like operator R in (1.5). In this case, E is called the coarse-grid operator. Combining those elements we can define after [48] the abstract additive coarse grid correction in form of following preconditioner:

$$\mathscr{P}_{as} = M^{-1} + ZE^{-1}Z^T, \tag{1.20}$$

Using the same ingredients we can define another interesting preconditioner. The balancing Neumann-Neumann preconditioner, which is well-known as a FETI algorithm in domain decomposition method. For symmetric systems it was proposed by Mandel in 1993 [42]. A more general form of abstract balancing preconditioner for non-symmetric systems reads [23] as follows:

$$\mathscr{P}_{bNtN} = Q_D M^{-1} P_D + ZE^{-1}Y^T, \tag{1.21}$$

where $E = Y^TAZ$, $P_D = \mathbb{1} - AZE^{-1}Y^T$, $Q_D = \mathbb{1} - ZE^{-1}Y^TA$. For SPD systems, by choosing $Y = Z$, the authors in [57] define similar preconditioner

$$\mathscr{P}_{a-def2} = Q_D M^{-1} + ZE^{-1}Z^T \tag{1.22}$$

which is as robust as $\mathscr{P}_{bNtN}$ but less expensive.

The subscript '*def*' in (1.22) states for 'deflation', which is another projection method. In deflation method, M^{-1} is often a traditional preconditioner, such as the Incomplete Cholesky factorization. Furthermore, the deflation subspace matrix, Z, consists of so-called deflation vectors, used in the deflation matrix P. In this case, the column space of Z spans the deflation subspace, i.e., the space to be projected, out of the residuals. It often consist of eigenvectors, or piecewise constant or linear vectors, which are strongly related to DDM. The good and simple example is a deflation subspace Z proposed by Nicolaides [47]:

$$(z_k)_l = \begin{cases} 1 & l \in \Omega_k \\ 0 & l \notin \Omega_k \end{cases}, \tag{1.23}$$

hence Z has the form

$$Z = \begin{pmatrix} \mathbb{1}_{\Omega_1} & 0 & \cdots & 0 \\ \vdots & \mathbb{1}_{\Omega_2} & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \mathbb{1}_{\Omega_J} \end{pmatrix},$$

where J is the number of subdomains Ω_j. If instead one chooses eigenvectors for building Z. The corresponding eigenvalues would be shifted to zero in the spectrum of the deflated matrix. This fact has motivated the name 'deflation method'.

1.5 Discussion

In the context of systems related with reservoir simulations, we have to deal with problems which coefficients have jumps of several orders of magnitude and are anisotropic (like in equation arising in porous media flow simulations through Darcy's law). In this case algebraic DDM (in form of preconditioner (1.8) combined with Krylov iterative method) suffers from plateaux in the convergence due to the presence of very small isolated eigenvalues in the spectrum of the preconditioned linear system [24]. Another weakness of DDM is the lack of a mechanism to exchange information between all subdomains in the preconditioning step, thus the condition number grow with the number of subdomains. To overcome those drawbacks we can improve classical DDM by two species: *enhancement of interface condition* and *global communication mechanism.*

The classical Schwarz method is based on Dirichlet boundary conditions and overlapping subdomains are necessary to ensure convergence. However we showed (see §1.2) that we can use more general interface condition in order to accelerate the convergence and to permit non overlapping decomposition. If exact absorbing condition are used (DtN maps) we have optimal solution in terms of iterations count, nevertheless they are practically very difficult to use because of their non-local nature. However we can apply some algebraic approximation (see algorithm 1) techniques and build small and local DtN maps.

In order to establish *global communication* between subdomains for classical DDM algorithms, we can solve a "coarse problem" with a few degrees of freedom per subdomain

in each iteration. Such methods are close in sprit to multigrid methods and especially to two-level methods which we have discussed in §1.4.

The first goal for this thesis consists in the research for algebraic approximation for optimal interface condition which can be used in form of the sub-block enhancement in modified Schwarz method (2.12) (see Chapter 3). Then we put an effort on defining suitable coarse space operator in order to construct algebraically an efficient two-level method (see Chapter 4).

Chapter 2

ADDM*lib*: Parallel Algebraic DDM Library

In this chapter, we discuss the design of ADDMlib[1]*, an object-oriented library written in modern C++ for the application of algebraic domain decomposition methods in the solution of large sparse linear systems on parallel computers.*

The main design goal for ADDMlib is to provide flexible framework for parallel solver developers which are developing new algorithms for algebraic domain decomposition. It is our intent that ADDMlib reflects its main purpose by providing a set of carefully designed objects grouped in four logical groups. The first group "*Interface for domain decomposition*" consists in necessary data structures and methods for managing data in distributed memory environment. The second group of classes is responsible for *communication* between distributed data and along with first group they make basis for implementation of third group which is a *linear algebra kernel*. Finally on top of it we have interface for domain decomposition algorithms which use all components from predefined groups. The simple schema of ADDMlib structure is presented on Figure 2.1. The composition of ADDMlib determines the structure of this chapter. At the outset we describe briefly the first three groups and then we present algorithms and techniques used in the proposed solution.

1. First version of library was originally written and developed by Frédéric Nataf, Pascal Havé and Serge van Criekingen.

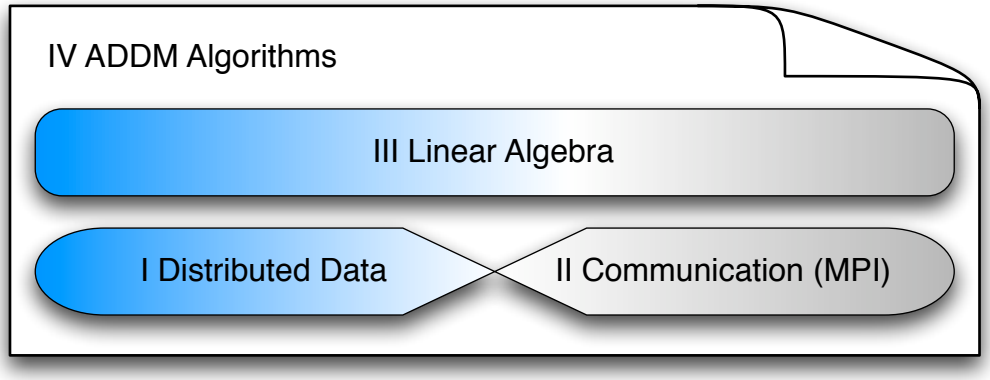


Figure 2.1: ADDM library structure.

2.1 Interface for Domain Decomposition and Communication

Since many types of parallel architectures exist in scientific computing (e.g., the "*shared memory*" or the "*distributed memory message-passing*" models), it is difficult to develop numerical libraries with data structures suitable for all of them. Thus in order to develop robust algorithms we need to specialize our library to a chosen architecture. The ADDMlib is designed for *Distributed Memory Architectures*.

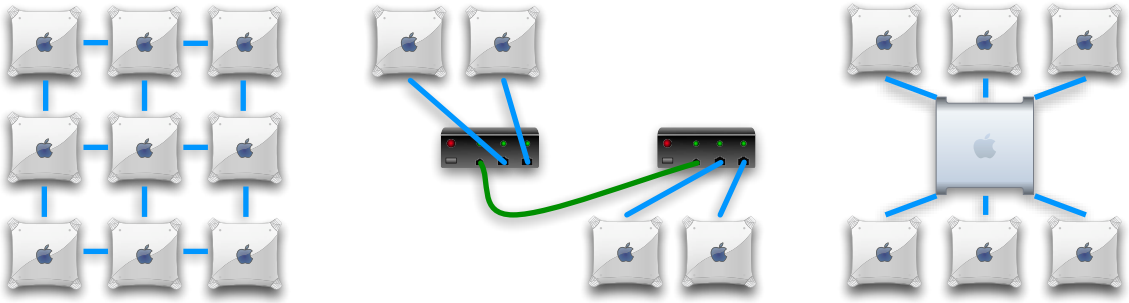
2.1.1 Distributed Memory Architectures

A typical distributed memory system consists of a large number of identical process with their own memories, which are interconnected in a regular topology (see Figure 2.2). In this case each processor unit can be viewed as a complete processor with its own memory, CPU and I/O [2] subsystem. In message-passing models there is no global synchronization of the parallel task. Instead, computation are *data driven* because a processor performs a given task only when the operands it requires become available. The programmer must program all the data exchange explicitly between processors.

Distributed memory computers can exploit locality of data in order to keep communication cost to minimum. Thus, a two-dimensional processor grid or three-dimensional hypercube grid, is perfectly suitable for solving discretized elliptic PDEs in DDM framework by assigning each subdomain to a corresponding processor. It is because the iterative methods for solving resulting linear system will require only interchange of data between adjacent subdomains.

Thanks to flexibility, the architecture of choice in nowadays is the distributed memory machine using message passing. There is no doubt that this is due to the availability of excellent communication software, such as MPI; see [32]. In addition, the topology is often

2. Input/Output



(a) Grid (b) Tree (c) Star

Figure 2.2: Diagram of different network topologies.

hidden from the user, so there is no need to code communication on specific configurations such as hypercubes or specific grids. Since this mode of computing has penetrated the application areas and industrial applications, it is likely to remain a standard for some time.

Multi-core Strategies

The introducing of multi-core processors to *High Performance Computing* (HPC) is often viewed as a huge benefit (more cores in the same space for the same cost). While number of cores is an advantage, multi-cores has introduced an additional layer of complexity for the HPC users. There are many new decisions that the programmer and end user must make in regards to multi-core technology.

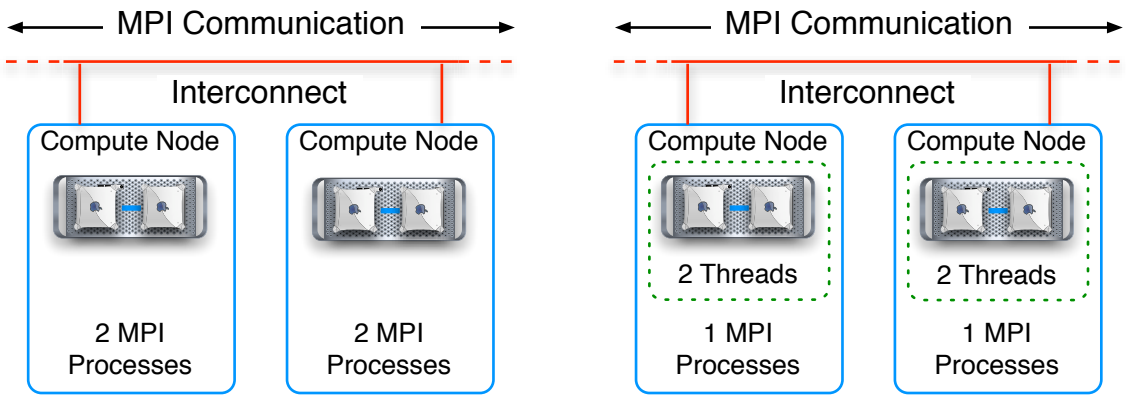
A multi-core processor looks the same as multi-socket single-core server to the operating system, thus programming in this environment is essentially a matter of using POSIX threads.[3] Thread programming can be difficult and error prone. Thus, *OpenMP* was developed to give programmers a higher level of abstraction and make thread programming easier. In threaded or OpenMP environment, communication happens through memory. A single program (process) will branch or launch multiple threads, which are then executed on separate cores in parallel. The entire program shares the same memory space i.e., there is no copied data, there is only one copy and it is shared between threads. In contrary to MPI, which basically copies memory and sends it between programs (process). Thus, MPI type of communication is best for distributed memory systems like clusters. The programs sending messages do not share memory. By design, the MPI process can be located either on the same server or on a separate servers. Regardless of where it runs, each MPI process has its own memory space from which messages are copied.

3. POSIX (**P**ortable **O**perating **S**ystem **I**nterface [for Uni**x**]) is the name of a family of related standards specified by IEEE to define the application programming interface (API), along with shell and utilities interfaces for software compatible with variants of the Unix operating system, although the standard can apply to any operating system.

As mentioned, MPI can run across distributed servers and on SMP (multi-core) servers, while OpenMP is best to run on a single SMP. For this reason, MPI codes usually scale to large number of servers, while OpenMP is restricted to a single operating system domain. There is one important assumption about OpenMP (which in some test appears to not be a true [1]), by the nature of threads, OpenMP is faster than distributed MPI programs (running on multiple servers) for the same number of cores (presumably because of the communication overhead introduced by MPI). Hence, recently popular approach to multi-core HPC is to use both MPI and OpenMP in the same program. For example, a traditional MPI program running on two 2-core servers (each server contains two two-core processors) is depicted on Figure 2.3(a). There are a total of four independent process for the MPI job. If one were to use both MPI and OpenMP, then strategy on Figure 2.3(b) would be the best way to create a hybrid program. As shown in the figure, each node runs just one MPI process, which then spawns two OpenMP threads.

Facing the multi-core problem during designing of ADDMlib, we decided to stick with classical MPI and strategy depicted on Figure 2.3(a). There is couple of reason for that. First, algorithms and data structures in ADDMlib are implemented in such a way, that amount of data to exchange is always minimized. Thus, OpenMP environment in which cores share memory, may not yield a great improvement. Second, ADDMlib intends to be library for solving big sparse linear systems, which scale is dedicated to larger numbers of servers (however, there is a product from Intel called *ClusterOpenMP* that can run OpenMP application across a cluster). Finally, the multi-core processors in *High Performance Computing* are challenge also for MPI library developers. Some of them try to adapt existing MPI libraries in such way that they will take advantages of heterogeneous architectures (multi-core and shared memory) [10]:

> "The increasing numbers of cores, shared caches and memory nodes within machines introduces a complex hardware topology. High-performance computing applications now have to carefully adapt their placement and behaviour according to the underlying hierarchy of hardware resources and their software affinities. We introduce the Hardware Locality (hwloc) software which gathers hardware information about processors, caches, memory nodes and more, and exposes it to applications and runtime systems in a abstracted and portable hierarchical manner. hwloc may significantly help performance by having runtime systems place their tasks or adapt their communication strategies depending on hardware affinities. We show that hwloc can already be used by popular high-performance OpenMP or MPI software. Indeed, scheduling OpenMP threads according to their affinities or placing MPI processes according to their communication patterns shows interesting performance improvement thanks to hwloc. An optimised MPI communication strategy may also be dynamically chosen according to the location of the communicating processes in the machine and its hardware characteristics."



(a) 4 way MPI program execution on two nodes (4 MPI processes).

(b) 4 way MPI-OpenMP program execution on two nodes (2 MPI processes, each with 2 OpenMP threads).

Figure 2.3: Hybrid approaches for parallel program execution.

2.1.2 Data Distribution in ADDM*lib*

Given a sparse linear system to be solved in a distributed memory environment, it is natural to map pairs of equation/unknowns to the same processor in a certain predetermined way. This mapping can be determined automatically by a graph partitioner or it can be assigned ad hoc from knowledge of problem (see §2.4 for future explanation and numerical experiments). Without any loss of generality we can introduce a following definition of partitioning:

Definition 2.1 (Partitioning)**.** *Let* D_Ω *be a domain, i.e., a set of unique indices that each correspond to an unknown. Let* $\{\mathrm{D}_{\Omega_\mathrm{I}}\}_{\mathrm{I}=1\ldots\mathrm{N}}$ *be a partition of* D_Ω*, i.e., a set of* N *disjoint subsets, such that* $\cup_{\mathrm{I}=1}^{\mathrm{N}} \mathrm{D}_{\Omega_\mathrm{I}} = \mathrm{D}_\Omega$.

In order to distribute linear system which originates from the discretization of a PDE on a domain Ω (see §1.1.1), we identify $\mathrm{D}_{\Omega_\mathrm{I}}$ with a unique `ID` and we call this structure (along with some additional information about overlaps; see §2.3) a `Part`. Each part is associated with a unique processor, therefore we can summarise that in ADDMlib the discretized domain D_Ω, decomposed into N subdomains , has its representation in N unique `Parts` and their associated data can be stored on p ($1 \le p \le \mathrm{N}$) processes.

A local data structure must be set up in each processor to allow basic operation, such as (global) matrix-vector product and preconditioning operations, to be performed efficiently. Hence global operator is divided into sparse `Partial Operators` which act on `Partial Vectors` i.e., components of global vector. For future explanation see §2.2.

It is important to preprocess the distributed data in order to facilitate the implementation of the communication tasks and to gain efficiency during the iterative process. The important observation is that a given process does not need to store information about global

structure of decomposed system. It is optimal to reduce information to neighboring `Parts` [4] since data exchange processes only through common interface between subdomains. Thus the preprocessing requires setting up for each process a "`Part Set`" object, which is a list of local `Parts` and `Part Infos` of their neighbors. Where `Part Info` is a container of essential and cheap to store, information about non local `Part`.

Since the `Part` is a fundamental constituent of domain decomposition realization in ADDMlib, the communication classes [5] implemented in ADDMlib support us in developing algorithms focused on data exchange between them, keeping informations about their physical distribution oven a cluster of processes hidden. In other words we can first decide what data should be exchange between given `Parts` (domain decomposition algorithm level) and then by using their `IDs` and ADDMlib interface for MPI we can easily establish data transfer between associated processes using point-to-point communication (data exchange between distributed memory).

2.2 Linear Algebra

The core object models for modern linear solver library consists of base classes capturing the mathematics, i.e., `Matrix`, `Vector`, `Iterative Solver` and `Preconditioner`. Their structures should be motivated by parallel architecture on which we plan to solve our problem and by operations we want to perform in order to obtain solution.

When we consider Krylov subspace techniques, namely, the preconditioned generalized minimal residual (GMRES) algorithm for the nonsymmetric case (or its flexible variant FGMRES), it is easy to notice that all Krylov subspace techniques require the same basic operations, thus the first step when implementing those algorithms on a high performance computer is identifying main operations that they require. We can list them after [51]:

- vector updates
- dot products
- matrix by vector multiplication
- preconditioner setup and operations.

For the sake of straightforward presentation, this section is divided into four logical parts; Vector, Matrix, Iterative Solver and Preconditioner. In each section emphasis is placed on concise description of data structures and algorithms used in implementation of associated elements and their basic algebra in ADDMlib.

4. In this case, neighboring in algebraic sense i.e., the two `Parts` are neighbour when they have a common interface. Thus process on which neighbours data structures are stored, can be physically very remote.

5. ADDMlib has object oriented interface for MPI (which is procedural in nature) in order to simplify and encapsulate typical communication tasks (e.g., sending a block of data to pointed process)

2.2.1 Vector (DDMVector)

Vector operations, such as linear combinations of vectors and dot products, are usually the simplest to implement on any computer. Operation of the form

$$y(1:n) = y(1:n) + a \cdot x(1:n),$$

where a is a scalar and $y, x \in \mathbb{R}^n$ two vectors, are known as *vector updates*. On shared memory computers, parallel version of this operation is usually automatically generated by compiler, but on distributed memory computers, some assumptions must be made about the way in which the vector are distributed. The main assumption is that the vectors x and y are distributed in the same manner among processors, meaning that indices of the components of any vector that are mapped to a given processor are the same. From previous section we know that this mapping is encapsulated in `Part` and `PartSet` implementation. Thus global vectors (`DDMVectors` hereafter) x and y are divided into N `Partial Vectors` with unique `ID` and a size equal to the size of the corresponding sub-domain, i.e., the number of unknowns in the corresponding `Part`. Moreover all `Partial Vectors` with the same `ID` are stored on the same process. In this case the vector update operation will be translated into N independent vector update, requiring no communication. Specifically, for all PV (`Partial Vectors`) on current process (we can have couple of `Parts` per process), this processor will simply execute a vector loop of the form

$$\mathrm{PV}_y(1:pvn) = \mathrm{PV}_y(1:pvn) + a \cdot \mathrm{PV}_x(1:pvn),$$

where pvn is the number of variables in the local `Partial Vector`. The example of `DDMVector` divided into three `Partial Vectors` is depicted on Figure 2.4 along with decomposed domain Ω. We assume that there is one `Part` per process.

Another essential vector operation in Krylov techniques is *dot product*. To be more specific, the distributed dot product operation should compute the inner product $cin = x^{\mathrm{T}} y$ of two distributed vectors (x, y) (`DDMVectors`) and then make the result cin available in each processor. Hence, this result is needed to perform vector updates or other operations in each node. For a large number of processors, this sort of operation can be very costly in terms of communication cost. Fortunately MPI provides global reduction operation, which is useful for global operations such as sums and global max-min calculations. This is a single routine which in effective way "collect" single values (in our case local $\sum \left(cin_{\mathrm{PV}} = \mathrm{PV}_x^{\mathrm{T}} \mathrm{PV}_y \right)$) from all processes involved in the computation and perform the chosen operation (add, max, min, multiply). Thus to obtain "global" dot product, we perform local operations over partial vectors, then we sum results locally and we pass output to MPI_Allreduce subroutine in order to obtain global sum (the cin value) available on all process.

2.2.2 Matrix (DDMOeprator)

The `Matrix` class has several uses. It is used in `Iterative Linear Solver` class to define the problem to solve. Its matrix-vector multiply (`axpy`) member function is used by

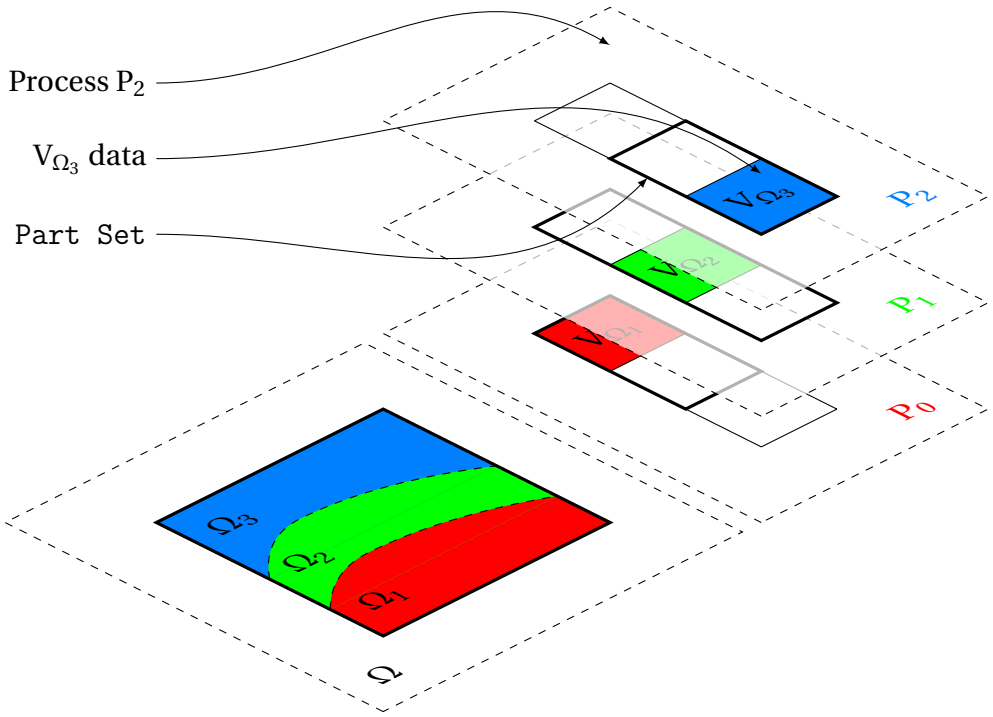


Figure 2.4: `DDMVector` structure and its division into `Partial Vectors` according to decomposition of domain Ω.

the `Krylov` class to implement Krylov-based algorithms (GMRES, FGMRES and BiCGSTAB). Subclasses of `Matrix` introduce access functions that provide abstractions for accessing the underlying data structures. Hence new interface conditions or preconditioners can be written in terms of these access functions. `Matrix` class in ADDMlib should be see then, as a *container* with some managing functionalities, which allows robust localisation of its distributed data structures associated with given subdomain in order to perform some algebraic operation over the global problem.

To indicate complex functionality, all object of type `Matrix` we will call `DDMOperators` hereafter.

Structure and Sparse Storage Formats

The computational kernels for performing sparse matrix operations such as matrix-vector products are strongly connected with the data structures used. Lets consider *Compressed Sparse Row* (CSR) format for instance. It consists of three arrays: an array A$(1:nnz)$ to store the nonzero elements of the matrix row-wise, an integer array JA$(j:nzz)$ to store the column positions of the elements in the real array A, and finally, a pointer array IA$(1:n+1)$, the i^{th} entry of which points to the beginning of the i^{th} row in the array A and JA. To perform the matrix-vector product $y = Ax$ in parallel using this format note (see algorithm 2) that each component of the resulting vector y can be computed independently as the dot product of the i^{th} row of the matrix with the vector x. On distributed memory architecture,

Algorithm 2 CSR Format - Matrix-Vector multiplication in dot product form

1: **for** $i = 1$ to n **do**
2: $k_1 = \mathrm{IA}(i)$
3: $k_2 = \mathrm{IA}(i+1) - 1$
4: $y(i) = dotproduct(\mathrm{A}(k_1 : k_2), x(\mathrm{JA}(k_1 : k_2)))$
5: **end for**

the outer loop can be split into a number of steps to be executed on each processor. From previous section we know that in order to distribute global vector (`DDMVector`) we divide it into `PartialVectors`. If we apply the same partition of unknowns on indices denoting rows and columns of global operator we will obtain block decomposition. Sub-blocks of original operator created in such a way are called `Partial Operators` hereafter. Each `Partial Operator` is a distinctive object which encapsulate its sparse data in a given format. For sake of simplicity each `Partial Opeartor` is denoted by pair (`PartInID`,`PartOutID`) in order to denote direction of its action (in terms of linear algebra). For example `ParialOperator`$(0,1)$ is an operator which columns indices belong to part with `ID` 0, while rows indices are mapped to part with `ID` 1, thus this operator acts on `Partial Vector` with `ID` 0, and the result of this operation (e.g., axpy) returns `Partial Vector` with `ID` 1 i.e., `Partial Vector` in size of `Part` with `ID` 1. Moreover `Partial Operators` whose `PartInID` is the same as its `ParOutID`, are *endomorphic* since the `Parts` they apply are the same as the `Parts` containing the result of their application (thus endomorphic `Partial Operators` are diagonal blocks). If `PartInID` and `PartOutID` are different, the corresponding `Partial Operator` is *exomporphic*.

The important observation is that *exomorphic* `Partial Operators` are more sparse then their *endomorphic* counterparts. It is due to fact that *exomorphic* operators have nonzeros only in rows whose associated unknowns liey on the interface of given subdomain. For that reason storage formats differ between diagonal and off-diagonal `Partial Operators`. Sparse data in *endomorphic* operators are stored in CSR format and for compress nonzeros in *exomorphic* operators we use *Compressed Sparse Rows & Columns Format* (CSRC) in which both, rows and columns storage data is compacted. CSRC format can easy be applied by adding additional two integer vectors to standard CSR format. Those extra vectors consist in the list of indices for which corresponding rows and columns has non zero values.

Notice that the unique bind of `PartID` and process `ID` puts storage constraints only on endomorphic operators i.e., since we associate chosen `Part` to a given process, it holds data structures with all `Partial Vectors` with the same `PartID`, thus it is preferable that endomorphic `Partial Operator` acting on it will be store on the same process, in contrary to exomorphic `Partial Operators` for which user can choose arbitrary distribution.

The example of global linear system and its decomposition into `Partial Vectors` and `Opearators`, along with its distribution among process, is depicted on Figure 2.5.

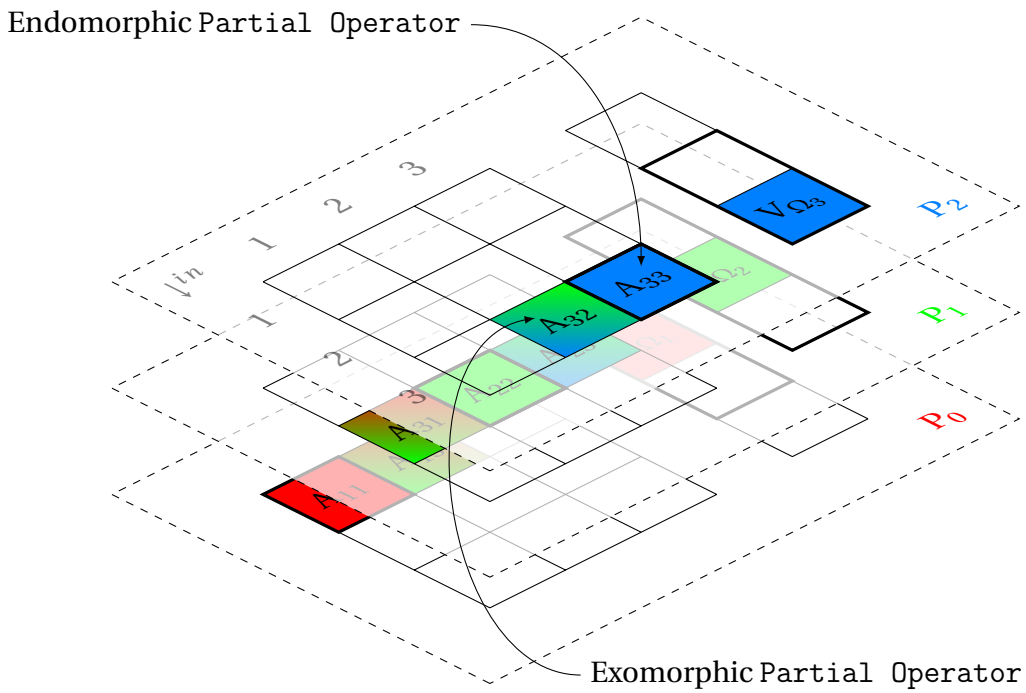


Figure 2.5: Decomposition of global linear system into Partial Vectors and Operators along with their distribution among three processes. Decomposition into subdomains refers to domain Ω depicted on Figure 2.4.

Matrix-Vector Product

To perform a global matrix-vector product, with Partial Operators encapsulated in DDMOperator, each processor must perform the following operations. First, multiply the local Partial Vectors by local endomorphic Partial Operators. Second, obtain the external variables from neighboring processor and their Partial Vectors in order to (third step) multiply these by local exomorphic Partial Operators and add the resulting vector to the one obtained from the first multiplication. Obviously the first and third step can be done in parallel.

2.2.3 Preconditioner

When we consider Krylov iterative techniques, vector updates, dot and matrix-vector product described in previous sections, are all we need to successfully implement iterative linear solver. There is a number of algorithms which use Krylov spaces in order approximate solution using this small set of algebraic operations. They differ in the restrictions or optimality conditions associated with the computed solution. In this report we focus only on the family [6] of **G**eneralised **M**inimum **RES**idual methods (GMRES) which basic algorithm

6. Left/Right preconditioned GMRES, flexible variant of GMRES (FGMRES) and GMRES with restarting [51, 31].

has been introduced by Yousef Saad and Martin H. Schultz in 1986 [52], for other techniques we refer the reader to the books [51, 31].

For many difficult problems Krylov iterative solvers may converge very slowly, or even diverge. The convergence of iterative methods can be improved by transforming general linear system AU = F into another system which is easier to solve. A preconditioner is a matrix that realises such a transformation. Thus, if M^{-1} is a non-singular matrix which approximates A^{-1}, then the transformed linear system:

$$M^{-1}AU = M^{-1}F, \tag{2.1}$$

might be solved faster. The system (2.1) is preconditioned from the left, but one can also precondition from the right side:

$$AM^{-1}t = F. \tag{2.2}$$

Once the solution t is obtained, the solution of the AU = F is recovered by $U = M^{-1}t$.

ADDM Preconditioning

Since ADDMlib intend to be a platform for applying a domain decomposition techniques, our choice of preconditioner is motivated by additive Schwarz procedure, which basic form is presented in Algorithm 3. The preconditioning matrix is simple to obtain from the additive

Algorithm 3 Additive Schwarz Iterations

Require: $D_\Omega = \cup_{i=1}^{N} D_{\Omega_i}$ thus $A_i = R_i A R_i^T$ (see §2.2.2)

1: **for** $i = 1$ to N **do**
2: Compute $\beta_i = R_i^T A_i R_i (F - AU)$
3: **end for**
4: $U_{new} = U + \sum_{i=1}^{N} \beta_i$

Schwarz procedure. For better picture let us introduce some notations:

Notation 2.1.

$$A_i = R_i A R_i^T \quad \texttt{PartialOperator}(i,i) \tag{2.3}$$
$$P_i = R_i^T A_i^{-1} R_i A \tag{2.4}$$
$$T_i = P_i A^{-1} = R_i^T A_i^{-1} R_i \tag{2.5}$$

For R_i *definition with example see §1.1.1 p. 20.*

Using new notation (N. 2.1), notice that the new iterate in ASM satisfies the relation

$$U_{new} = \left(\mathbb{1} - \sum_{i=1}^{N} P_i \right) U + \sum_{i=1}^{N} T_i F.$$

Thus, this iteration corresponds to a fixed-point iteration $U_{new} = GU + f$, with

$$G = I - \sum_{i=1}^{N} P_i, \quad f = \sum_{i=1}^{N} T_i F.$$

With the relation $G = \mathbb{1} - M^{-1}A$ between G and the preconditioning matrix M, the result is that

$$M^{-1}A = \sum_{i=1}^{N} P_i$$

and

$$M^{-1} = \sum_{i=1}^{N} P_i A^{-1} = \sum_{i=1}^{N} T_i.$$

Now the procedure for applying the preconditioned operator M^{-1} becomes clear (see Algorithm 4). Note that the **do** loop can be performed in parallel. Line 6 sums up the vec-

Algorithm 4 Additive Schwarz Preconditioner

1: Input: (`DDMVector`) v
2: Output: (`DDMVector`) $z = M^{-1}v$
3: **for** $i = 1$ to N **do**
4: Compute (`Partial Vector`) $z_i := T_i v = \left[\text{endomorphic } \texttt{PartialOperator}(i,i)\right]^{-1} v_i$
5: **end for**
6: Compute $z := z_1 + z_2 + \ldots + z_N$

tors z_i in each domain to obtain global vector z. Thus in ADDMlib vector z_i corresponds to `Partial Vector` with `ID`(i) and matrix T_i agrees with endomorphic `Partial Operator` with the same `PartID`.

From technical point of view, instead of "inverting" endomorphic operators, the preconditioner class in ADDMlib, provide an interface to external solvers like `SuperLU` [7] or `PETSc` library [8] in order to solve the following system

$$\left[\texttt{PartialOperator}(i,i)\right] z_i = v_i$$

By default we use a direct solver (*SuperLU*), but since we can also use *PETSc*, the number of available techniques for solving this local linear system is greatly extended by iterative methods like Boomer - Algebraic Multi-grids [6]. This successful "alliance" of ADDMlib with the external solvers, makes it a very flexible hybrid solver.

2.3 Overlaps

As it has been shown in previous chapter, the original Schwarz method need overlapping subdomains to converge. Moreover the slowness of the method and the need of the

7. SuperLU is a general purpose library for the direct solution of large, sparse, nonsymmetric systems of linear equations; see [18].
8. Portable, Extensible Toolkit for Scientific Computation; see [6, 7, 8].

overlap is linked (see Theorem 1.1). In order to increase the efficiency we introduce here "inflation" - the tool to create overlapping regions by duplicating unknowns at the algebraic level. Once an inflation has been performed, simple Dirichlet-type interface conditions can be enhanced by introducing more complex interface conditions. This leads to the modified Schwarz method, investigated in §2.5.

Definition 2.2 (Inflation)**.** *Let* D_Ω *be a domain, i.e. a set of unique indices that each correspond to an unknown. Let* $\{D_{\Omega_I}\}_{I=1...N}$ *be a partition of* D_Ω*, i.e. a set of* N *disjoint subsets, such that* $\cup_{I=1}^{N} D_{\Omega_I} = D_\Omega$*. Each subset* D_{Ω_I} *can be inflated into* $\widetilde{D}_{\Omega_I}$ *as follows. Let* $i_0, i_1, \ldots, i_{N_I}$ *denote the indices of* D_{Ω_I}*. Then* $\widetilde{D}_{\Omega_I}$ *is constructed by adding to* D_{Ω_I} *the duplication of the indices* j_β *belonging to any subset* $D_{\Omega_{J\neq I}}$ *such that*

Inflation

$$\widetilde{D}_{\Omega_I} = D_{\Omega_I} \cup \left\{ j_\beta \in D_{\Omega_J} \;\middle|\; \exists i_\alpha \in D_{\Omega_I},\quad a_{i_\alpha j_\beta} \neq 0 \right\} \tag{2.6}$$

where $a_{i_\alpha j_\beta}$ *is an element in matrix* A.

The inflated set of subscripts D_Ω^{infl} is in turn defined as the union

$$\widetilde{D}_\Omega = \cup_{I=1}^{N} \widetilde{D}_{\Omega_I} \quad \text{where} \quad D_{\Omega_I} \subset \widetilde{D}_{\Omega_I}. \tag{2.7}$$

In the sequel, we will often denote a set of indices with the same notation than its capitalised identifier, e.g. D_{Ω_I} by I and $\widetilde{D}_{\Omega_I}$ by $\widetilde{I}$. Finally, the inflated operator A^{inf} (or $\widetilde{A}$) is obtained by duplicating the appropriate rows and columns as we can see in following example.

2.3.1 The two domain case

For the sake of simplicity, consider the one dimensional, algebraically non-overlapping case presented on figure 2.6. When this problem is discretized, its forms the following linear, block system

$$\left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \left[\begin{array}{c} U_1 \\ \hline U_2 \end{array}\right] = \left[\begin{array}{c} F_1 \\ \hline F_2 \end{array}\right]. \tag{2.8}$$

Let us consider now an one level[9] inflation, the A_{11} block is then inflated by the non-zero elements in A_{12}, while the block A_{22} by non-zeros in A_{21}. Re-ordering the unknowns if necessary, we have

$$U_1 = \left[\begin{array}{c} U_{1_i} \\ \hline U_{1_\Gamma} \end{array}\right] \quad \text{such that} \quad F_1 = \left[\begin{array}{c} F_{1_i} \\ \hline F_{1_\Gamma} \end{array}\right],$$

$$U_2 = \left[\begin{array}{c} U_{2_i} \\ \hline U_{2_\Gamma} \end{array}\right] \quad \text{such that} \quad F_1 = \left[\begin{array}{c} F_{1_i} \\ \hline F_{1_\Gamma} \end{array}\right].$$

9. It is possible to execute inflation algorithm couple of times which quota we name "level" or "depth".

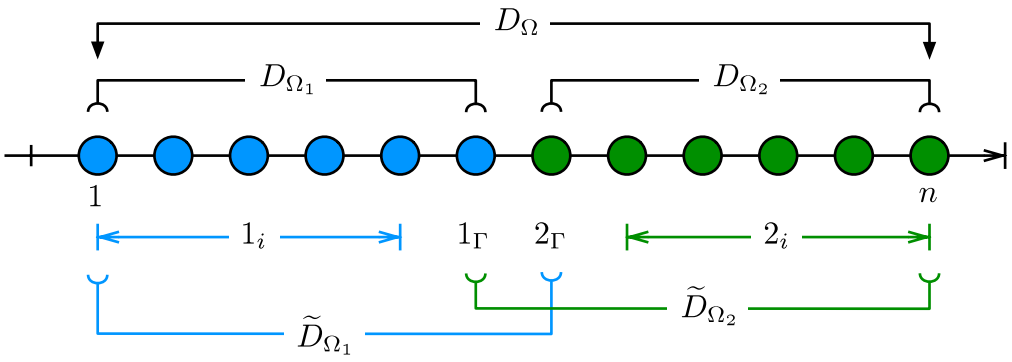


Figure 2.6: An 1D example of two subdomains inflation.

We then introduce the corresponding splitting for sub-operators

$$\mathrm{A}_{11} = \begin{bmatrix} \mathrm{A}_{1_i1_i} & \mathrm{A}_{1_i1_\Gamma} \\ \mathrm{A}_{1_\Gamma1_i} & \mathrm{A}_{1_\Gamma1_\Gamma} \end{bmatrix} \quad , \quad \mathrm{A}_{12} = \begin{bmatrix} \mathrm{A}_{1_i2_\mathrm{I}} & \mathrm{A}_{1_i2_\Gamma} \\ \mathrm{A}_{1_\Gamma2_i} & \mathrm{A}_{1_\Gamma2_\Gamma} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & \mathrm{A}_{1_\Gamma2_\Gamma} \end{bmatrix}$$
$$\mathrm{A}_{22} = \begin{bmatrix} \mathrm{A}_{2_i2_i} & \mathrm{A}_{2_i2_\Gamma} \\ \mathrm{A}_{2_\Gamma2_i} & \mathrm{A}_{2_\Gamma2_\Gamma} \end{bmatrix} \quad \text{and} \quad \mathrm{A}_{21} = \begin{bmatrix} \mathrm{A}_{2_i1_i} & \mathrm{A}_{2_i1_\Gamma} \\ \mathrm{A}_{2_\Gamma1_i} & \mathrm{A}_{2_\Gamma1_\Gamma} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & \mathrm{A}_{2_\Gamma1_\Gamma} \end{bmatrix}$$

such that equation (2.8) can be re-written in following form

$$\left[\begin{array}{cc|cc} \mathrm{A}_{1_i1_i} & \mathrm{A}_{1_i1_\Gamma} & 0 & 0 \\ \mathrm{A}_{1_\Gamma1_i} & \mathrm{A}_{1_\Gamma1_\Gamma} & 0 & \mathrm{A}_{1_\Gamma2_\Gamma} \\ \hline 0 & 0 & \mathrm{A}_{2_i2_i} & \mathrm{A}_{2_i2_\Gamma} \\ 0 & \mathrm{A}_{2_\Gamma1_\Gamma} & \mathrm{A}_{2_\Gamma2_i} & \mathrm{A}_{2_\Gamma2_\Gamma} \end{array}\right] \left[\begin{array}{c} \mathrm{U}_{1_i} \\ \mathrm{U}_{1_\Gamma} \\ \hline \mathrm{U}_{2_i} \\ \mathrm{U}_{2_\Gamma} \end{array}\right] = \left[\begin{array}{c} \mathrm{F}_{1_i} \\ \mathrm{F}_{1_\Gamma} \\ \hline \mathrm{F}_{2_i} \\ \mathrm{F}_{2_\Gamma} \end{array}\right]. \tag{2.9}$$

The subscripts "Γ" and "i" in introduced notation denote whenever the nodes corresponding to marked set of indices are located on interface or in interior of the subdomain.

We can now proceed an inflation. The inflation duplicates the unknowns k_Γ, thus we obtain after inflation

$$\left[\begin{array}{ccc|ccc} \mathrm{A}_{1_i1_i} & \mathrm{A}_{1_i1_\Gamma} & 0 & 0 & 0 & 0 \\ \mathrm{A}_{1_\Gamma1_i} & \mathrm{A}_{1_\Gamma1_\Gamma} & \mathrm{A}_{1_\Gamma2_\Gamma} & 0 & 0 & 0 \\ 0 & \mathrm{A}_{2_\Gamma1_\Gamma} & \mathrm{A}_{2_\Gamma2_\Gamma} & \mathrm{A}_{2_\Gamma2_i} & 0 & 0 \\ \hline 0 & 0 & 0 & \mathrm{A}_{2_i2_i} & \mathrm{A}_{2_i2_\Gamma} & 0 \\ 0 & 0 & 0 & \mathrm{A}_{2_\Gamma2_i} & \mathrm{A}_{2_\Gamma2_\Gamma} & \mathrm{A}_{2_\Gamma1_\Gamma} \\ \mathrm{A}_{1_\Gamma1_i} & 0 & 0 & 0 & \mathrm{A}_{1_\Gamma2_\Gamma} & \mathrm{A}_{1_\Gamma1_\Gamma} \end{array}\right] \left[\begin{array}{c} \mathrm{U}_{1_i} \\ \mathrm{U}_{1_\Gamma} \\ \mathrm{U}_{2_\Gamma} \\ \hline \mathrm{U}_{2_i} \\ \mathrm{U}_{2_\Gamma} \\ \mathrm{U}_{1_\Gamma} \end{array}\right] = \left[\begin{array}{c} \mathrm{F}_{1_i} \\ \mathrm{F}_{1_\Gamma} \\ \mathrm{F}_{2_\Gamma} \\ \hline \mathrm{F}_{2_i} \\ \mathrm{F}_{2_\Gamma} \\ \mathrm{F}_{1_\Gamma} \end{array}\right]. \tag{2.10}$$

2.3.2 Implementation

In order to demonstrate how the inflation was implemented in the ADDMlib, we divided this subsection into four parts according to distinct actions which the inflation algorithm needs to perform. For each step, the graphical schema has been constructed. All steps are depicted on Figure 2.7.

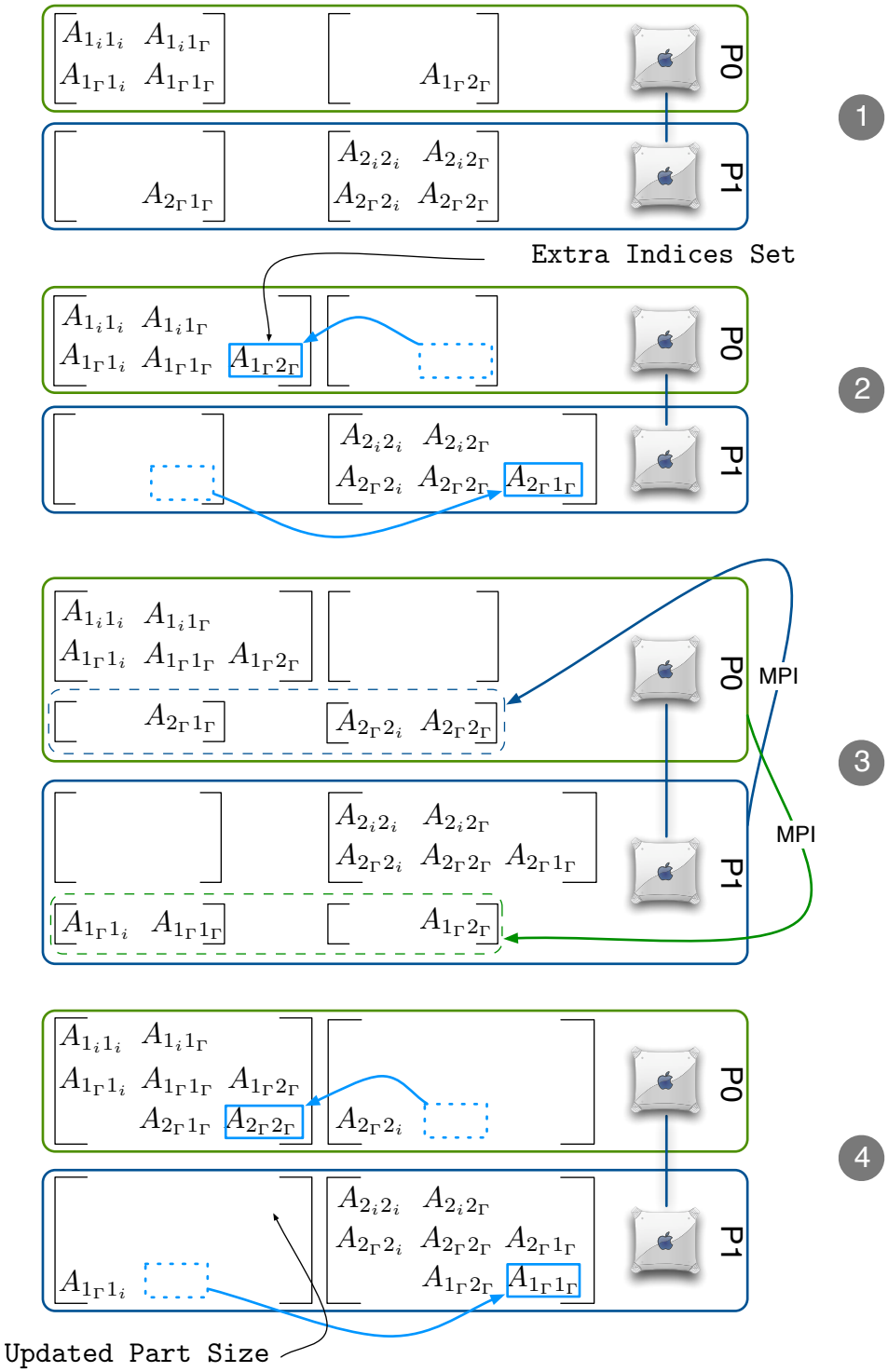


Figure 2.7: Four main steps of Inflation algorithm implemented in ADDMlib.

First step - Reconfiguring Data Distribution From the §2.2 we know that exomorphic *Partial Operators* in ADDMlib can be distributed among processors without any restrictions. Hoverer for inflation process it is preferable that all *Partial Operators* with the same *PartOutID* (see §2.2.2) are stored on the same process. In such configuration each process has instant access to full rows of global system associated with unknowns which are mapped to it via *Part* interface. Thus (if needed) we change distribution of exomorphic operators in order to obtain convenient local disposition of *Partial Operators*.

Second step - *Extra Indices Set* Next for each *Part*, algorithm collect all columns in exomorphic *PartialOperators* which has non-zero values i.e., collected columns are extracted from all *Partial Operators* with the same *PartOutID* but different *PartInIDs*. After that, the extracted data is added to endomorphic *PartialOperator*, thus proceeded *Part* is extended by collected in this way column indices i.e., new unknowns from neighboring subdomain interfaces. In order to keep information about origin of those additional indices, the new data structure is created for each inflated *Part*; the *Extra Indices Set*.

Third step - Rows Duplication In third step, for all entries in *Extra Indices Set* the inflation algorithm collect corresponding rows of global operator i.e., for each extra index we know its origin *Part ID* (*PartInID* of *Partial Operator* from which we have extracted corresponding columns), thus we need to duplicate row with the same index from all *Partial Operators* with *PartOutID* equal to proceed index origin *Part ID*. This sparse data corresponds to whole rows of global operator.

Fourth step - *Part Set update* Duplicated rows are added to the inflated *Partial Operators* in such a way that non-zeros for which column indices are now in the *Extra Indices Set* are moved to corresponding, new positions in the endomorphic *Partial Operator*. Non-zeros in positions which indices do not belong to the inflated *Part*, are new entries in the exomorphic *Partial Operators*. When all data is correctly associated, the *Part Set* on each processor is updated by new sizes of the inflated *Parts*

To inflate the `DDMVector`, we simply use `Extra Indices Sets` created during inflation of the `DDMOperator`. Thus, each `Partial Vector` is extends by duplicated values, pointed by entries in corresponding set of `Extra Indices`.

The inflation process can be repeated many times. The number of repetition we name "inflation level" or "inflation depth" hereafter.

2.3.3 Numerical experiments

In order to present influence of overlap on ASM performance, we can process the following numerical experiment:

Numerical Experiment 2.1. *Let us consider Laplace's equation* $-\Delta(\mathbf{u}) = 0$*, discretized using P1-type finite elements on 2D unit square in size* $\mathrm{N}_x \times \mathrm{N}_y$ *and triangulated by the* Delaunay-Voronoi*-type algorithm. On left and right side of the square we pose Dirichlet condition* $u = 0$ *and Neumann condition* $\frac{\partial u}{\partial \vec{n}} = 0$ *on top and bottom. The righthand side of resulting linear*

system is a function f which gives random values from set $\langle 1,2)$ (fixed for all variants of test). For domain decomposition we use decomposition into $M_x \times M_y$ subdomains, each of size $n_x \times n_y$ (thus, $N_x = n_x M_x$ and $N_y = n_y M_y$). We solve the resulting discrete system using GMRES left preconditioned by ASM. The initial guess is chosen to be $\mathbf{u}^{(0)} = 0$ (if experiment description say no different) and the stopping criterion $\|r_i\| \leq tol \cdot \|r_0\|$ for $tol = 1 \times 10^{-6}$. The estimated condition number is given as $\kappa_{\approx} = \lambda_{max}/\lambda_{min}$ where $\lambda_{\{min,max\}}$ are the approximated, extreme eigenvalues of $(M^{-1}A)$.

Influence of the inflation depth on condition number (κ) and number of iterations in experiment 2.1 is presented in table 2.1. For the experiment we set $n_x = n_y = 50$ and $M_x = M_y = 4$.

Method	$\kappa_{\approx}$	*n-iter*
ASM + no inflation	317.94	39
ASM + 1 level inflation	104.86	27
ASM + 2 level inflation	61.98	21
ASM + 3 level inflation	43.56	17

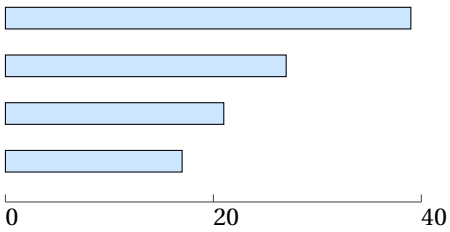


Table 2.1: Influence of the inflation depth on condition number (κ) and number of iterations (n-iter) in ASM method.

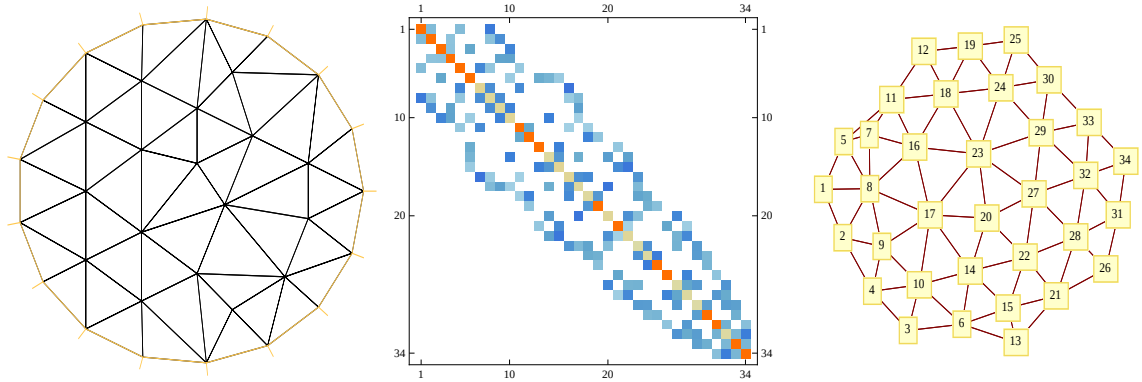
2.4 Partitioning with weights

The primary problem that a programmer needs to face when solving a problem on a parallel computer is to decide how to subdivide and map data into processors. Distributed memory computers allows a mapping of the data in arbitrary fashion but this automatically creates question how to find a good mapping. Thus efficient techniques must be available for partitioning an arbitrary graph.

From basic definition for a general sparse linear system whose adjacency graph is G = (V, E), the k-way graph partitioning problem is defined as follows: given a graph G = (V, E) with $|V| = n$, partition V into k subsets, $V_1, V_2, \ldots, V_k$ such that $V_i \cap V_j = \emptyset$ for $i \neq j$, $|V_i| = n/k$, and $\cup_i V_i = V$, and the number of edges of E whose incident vertices belong to different subset is minimized. A k-way partition of V is commonly represented by a partition vector P of length n, such that for every vertex $v \in V$, $P[v]$ is an integer between 1 and k, indicating the partition at which vertex v belongs [10].

The underlying goal of adjacency graph partitioner is to achieve a good load balance of the work among the processors as well as ensure that the ratio of communications to computation is small for the given task. There are a number of available software for graph parti-

10. This is a way how we pass partition to ADDMlib in order to create `Parts`



(a) Example of the meshed domain (b) Sparsity of the operator originate from discretized (P1-FEM) problem on mesh *a*. (c) Adjacency graph of the matrix depicted on figure 2.8(b)

Figure 2.8: Discrete components of PDE solution.

tioning. The two most popular; METIS [36] and SCOTCH [16] present very good performance keeping good quality of partition (mostly in terms of data balance).

However, there are different ways to partition the computational domain. For instance when we have access to the mesh on which we solve our problem, we can use availability of geometrical informations in order to perform decomposition (we call it manual partitioning hereafter).

During our experiments we quickly noticed that the way how adjacency graph is partitioned has strong influence on overall performance of algebraic domain decomposition methods which we use in ADDMlib. Because adjacency graph partition is independent of values in underlying sparse matrix (without user's tricks), in order to obtain different partition we were changing the automatic partitioning algorithm, or we were using manual partitioning.

There is certain number of problems for which clever partitioning in domain decomposition methods can increase robustness of the iterative process. For example, for highly anisotropic problems it is preferable to keep interfaces of subdomains along anisotropy direction, otherwise (e.g., in extreme case when interfaces goes perpendicularly to anisotropy direction) occurrence of big jumps of coefficients along interface can slow down convergence of iterative method. Generally speaking, it is preferable to keep together all nodes which are strongly connected and disengage (to form subdomain) those which are connected weakly. Therefore we can ask a following question:

> *Is it possible to extract algebraically some information about physical properties of the problem to solve, and use them to obtain better partition ?*

The k-way partitioning problem can be naturally extended to graphs that have weights associated with the vertices and the edges of the graph. In this case, the goal is to partition the vertices into k disjoint subsets such that the sum of the vertex-weights in each subset is the same, and the sum of the edge-weights whose incident vertices belong to different subset is minimised. Therefore we can "smuggle" some physical properties of underlying system

by defining custom weights for all edges in graph. When we relate graph partition mechanism i.e., cutting graph edges, to highly anisotropic problem, the way how we should define values of the edges weights comes naturally. In order to avoid partition with interfaces of the subdomains going along a direction of anisotropy we need to associate edges parallel to this direction with big weight. Thus from the minimisation process point of view, it will be a big "cost" for partitioner to cut them, in contrary to edges which are oriented perpendicularly to anisotropy direction which we associate with small weights in such way that it will be "cheap" to cut them. However, we do not need to know the geometrical orientation of edges to compute for them suitable weights. Some simple calculation inherited from algebraic multi grids techniques [56], express the desired properties described above. Thus, since number of edges of the adjacency graph is equal to the number of non-zeros in underlying sparse matrix, we can easily compute edge weight using values of underlying matrix via following formula

Automatic weight labelling

$$c = \left\lfloor \left(\frac{|a_{ij}|}{|a_{ii}| + |a_{jj}|} \times \gamma_{const} \right) \right\rfloor, \tag{2.11}$$

where $\lfloor x \rfloor$ is the floor function maps a real number to the next smallest integer and γ_{const} is an arbitrary constant.

As we will show in following subsections, the adjacency graph partitioned with such defined weights on edges, increase robustness of algebraic domain decomposition methods used in ADDMlib.

2.4.1 Implementation

It is common policy that graph partitioners take as input the adjacency structure of the graph and the weights of the vertices and edges (if any) stored using the CSR format. In this format adjacency structure of a graph with n vertices and m edges is represented using two arrays, namely $xadj$ and $adjncy$. The $xadj$ array is of size $n+1$ whereas the $adjncy$ array is of size $2m$ (this is because for each edge between vertices v and u we store both (v,u) and (u,v)). The weights of the edges are stored in an additional array, namely $adjwgt$, which contains $2m$ elements in such way that the weight of the edge $adjncy[j]$ is stored at location $adjwgt[j]$. The edge weights are most often integers due to performance of computing.

Since the CSR format is a standard way to store the adjacency structure of the graph, we can easily extract it from the sparse matrix stored in the same format. When we take a closer look on the CSR storage format used in ADDMlib (see §2.2.2 on page 41) we notice that is enough to pass as input to partitioner two arrays; IA$(1 : n+1)$ and JA$(1 : nzz)$ in order to obtain partition vector P$(1 : n)$.

To accelerate computation of the edges weights, we can copy from global operator all values laying on diagonal in order to create additional array A$diagval(1 : n)$. Thus, when

we proceed calculation via (2.11), we simply proceed through all non-zeros of the global operator (edges of adjacency graph) stored in CSR format (A$(1:nzz)$ array), in order to extract $|a_{ij}|$ values. Hence, thanks to A$diagval(1:n)$ array, we have instant access to values $|a_{ii}|$ and $|a_{jj}|$, necessary to complete computation of custom weights. Resulting additional array (A$weights(1:nnz)$) is an additional input to standard METIS/SCOTCH routines.

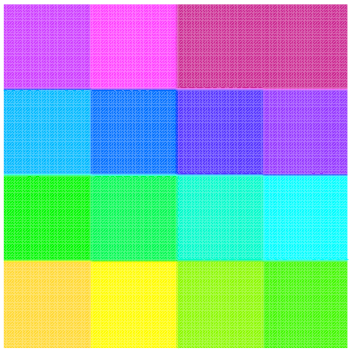
2.4.2 Numerical Experiments

In this subsection we present results of three numerical experiments under some common assumptions:

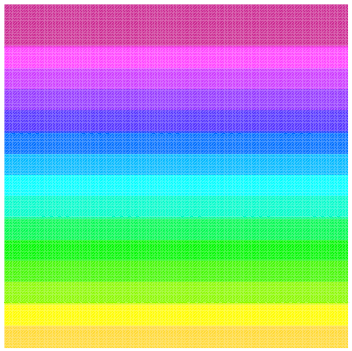
- All problems which associated linear system we solved, originates from P1-FEM discretization.
- For each experiment, the size of the problem (nodes of the two dimensional mesh) is fixed. Thus, only the way how the adjacency graph was partitioned varies.
- The resulting partitioned, linear systems, were solved using GMRES, preconditioned (left preconditioner) by ASM with following parameters
  - The initial guess is chosen to be $U^{(0)} = 0$.
  - The stopping criterion $\|r_i\| \leq tol \cdot \|r_0\|$ for $tol = 1 \times 10^{-6}$.
  - The roughly estimated condition number [11] is given as $\kappa_{\approx} = \lambda_{max}/\lambda_{min}$ where $\lambda_{\{min,max\}}$ are the approximated, extreme eigenvalues of $(M^{-1}A)$.

11. See remark 5.1 p. 118

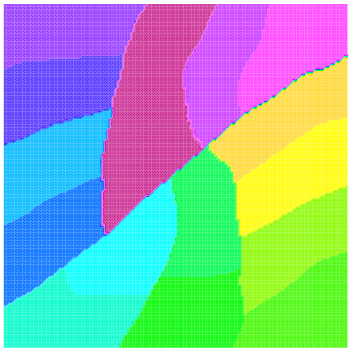
Numerical Experiment 2.2 (Laplace problem)**.** *Let us consider Laplace's equation* $-\eta\Delta(\mathbf{u}) = 0$ *discretized on 2D unit square in size* $N_x \times N_y$*, where* $N_x = N_y = 128$*. For domain decomposition we used six different configurations depicted below.*



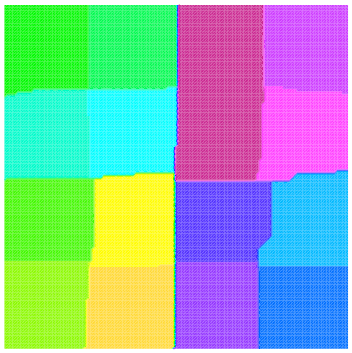
(a) Manual decomposition into 4×4 squares, each of size 32×32



(b) Manual decomposition into 16 strips, each of size 128×8

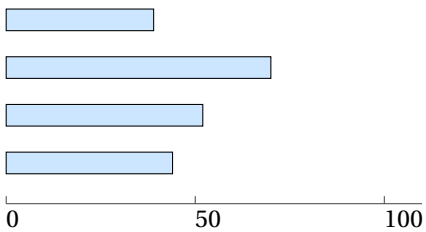


(c) SCOTCH graph partitioner without weights, for decomposition into 16 subdomains



(d) SCOTCH graph partitioner with weights define by (2.11). Decomposition into 16 subdomains.

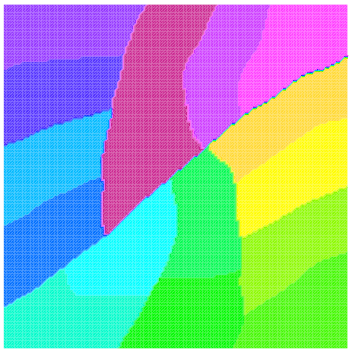
Paritioner	$\kappa_{\approx}$	*n-iter*
(a) Manual	214.98	39
(b) Manual	524.65	70
(c) SCOTCH	282.63	52
(d) SCOTCH + W	215.35	44



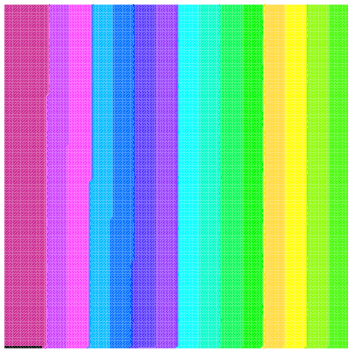
Numerical Experiment 2.3 (Anisotropy)**.** *Let us consider following, anisotropic problem:* $-\kappa\Delta(\mathbf{u}) = f$*, discretized on 2D unit square in size* $N_x \times N_y$*, where* $N_x = N_y = 128$ *and*

$$\kappa = \begin{bmatrix} \kappa_{xx} & 0 \\ 0 & \kappa_{yy} \end{bmatrix} = \begin{bmatrix} 1 \times 10^{-6} & 0 \\ 0 & 1 \end{bmatrix}.$$

For manual domain decomposition we use the same variants of partitioning as in Experiment (2.2)*.*

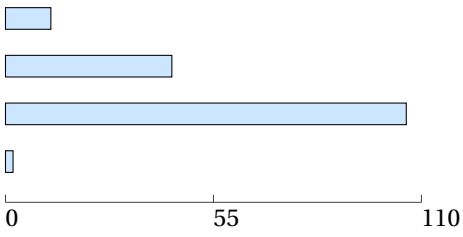


(e) SCOTCH graph partitioner without weights, for decomposition into 16 subdomains

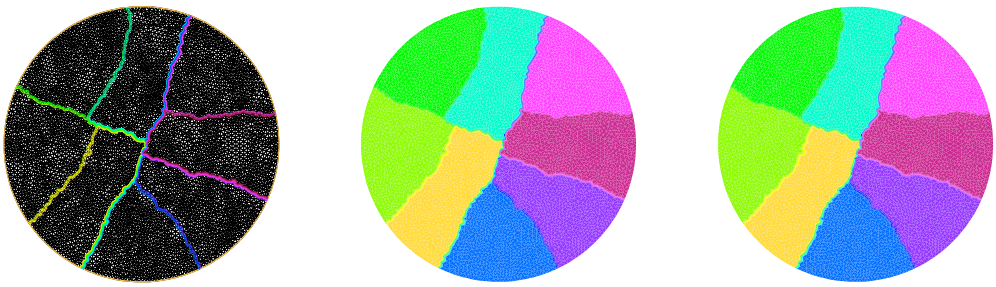


(f) SCOTCH graph partitioner with weights define by (2.11). Decomposition into 16 subdomains.

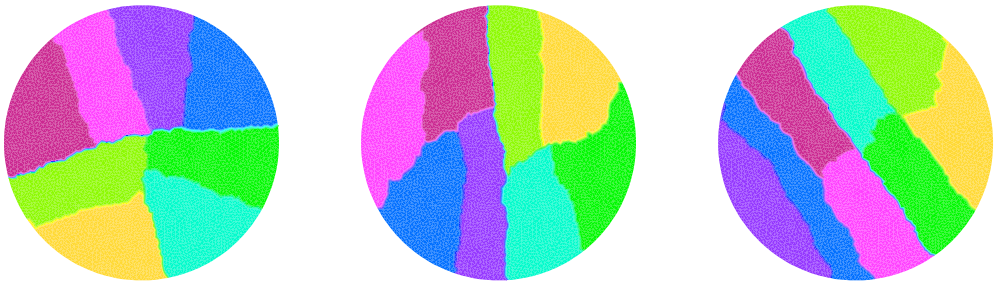
Paritioner	$\kappa_{\approx}$	*n-iter*
(a) Manual	217.19	12
(b) Manual	1048.51	44
(e) SCOTCH	342.72	106
(f) SCOTCH + W	1.00	2



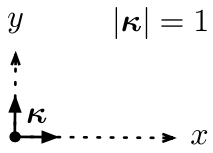
Numerical Experiment 2.4 (Anisotropy problem on unit disk)**.** *Consider anisotropic problem from Experiment 2.3, discretized on 2D unit disk, triangulated by the Delaunay-Voronoi-type algorithm. For domain decomposition we used only SCOTCH partitioner with arbitrary number of subdomains $n = 8$. The anisotropy tensor κ variants are depicted below.*

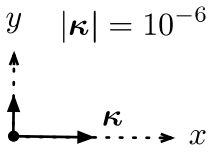


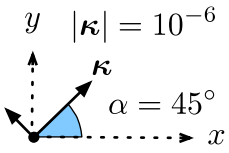
(a) (c) (e)



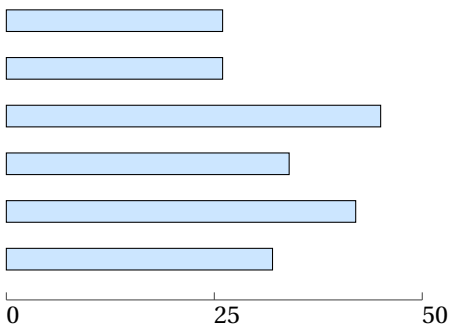
(b) (d) (f)







Paritioner	$\kappa_{\approx}$	*n-iter*
(a) SCOTCH	37.74	26
(b) SCOTCH + W	36.34	26
(c) SCOTCH	49.56	45
(d) SCOTCH + W	28.30	34
(e) SCOTCH	46.76	42
(f) SCOTCH + W	26.54	32



2.5 Modified Schwarz Method (MSM)

As it has been pointed out in §1.2, the main idea of the modified Schwarz method consists in using minimum overlapping along with interface conditions enhancement, in such a way that more than "Dirichlet data" is passed from one subdomain to another during the iterative process. In algebraic terms, these interface condition enhancements become additional sub-block matrices in the inflated matrix and that will be called "interface blocks".

2.5.1 The two sub-domains

Lets take equation (2.10) for which we will introduce the interface blocks S_1 and S_2 in order to obtain modified system

$$\left[\begin{array}{ccc|ccc} A_{1_i1_i} & A_{1_i1_\Gamma} & 0 & 0 & 0 & 0 \\ A_{1_\Gamma1_i} & A_{1_\Gamma1_\Gamma} & A_{1_\Gamma2_\Gamma} & 0 & 0 & 0 \\ 0 & A_{2_\Gamma1_\Gamma} & A_{2_\Gamma2_\Gamma}+S_1 & A_{2_\Gamma2_i} & -S_1 & 0 \\ \hline 0 & 0 & 0 & A_{2_i2_i} & A_{2_i2_\Gamma} & 0 \\ 0 & 0 & 0 & A_{2_\Gamma2_i} & A_{2_\Gamma2_\Gamma} & A_{2_\Gamma1_\Gamma} \\ A_{1_\Gamma1_i} & -S_2 & 0 & 0 & A_{1_\Gamma2_\Gamma} & A_{1_\Gamma1_\Gamma}+S_2 \end{array}\right] \left[\begin{array}{c} U_{1_i} \\ U_{1_\Gamma} \\ U_{2_\Gamma} \\ \hline U_{2_i} \\ U_{2_\Gamma} \\ U_{1_\Gamma} \end{array}\right] = \left[\begin{array}{c} F_{1_i} \\ F_{1_\Gamma} \\ F_{2_\Gamma} \\ \hline F_{2_i} \\ F_{2_\Gamma} \\ F_{1_\Gamma} \end{array}\right]. \tag{2.12}$$

The additive Schwarz method can be applied to solve this last system in parallel.

Remark 2.1 (Optimal choice for two domain case)**.** *The choice of* S_1 *and* S_2 *in* (2.12) *can be "adjusted" in such a way that Schur complements appears, i.e., taking*

$$S_1^{opt} = -A_{2_\Gamma2_i}A_{2_i2_i}^{-1}A_{2_i2_\Gamma} \tag{2.13a}$$

$$S_2^{opt} = -A_{1_\Gamma1_i}A_{1_i1_i}^{-1}A_{1_i1_\Gamma} \tag{2.13b}$$

is optimal, and the ASM in form of preconditioner in an iterative Krylov solver, converges for the system (2.12) *in two steps. Interface operators in such form we call* "optimal interface conditions" *(see §1.3).*

2.5.2 The three sub-domains case

In order to present how we can apply new interface conditions in more general case. Lets consider linear system (2.14), which applies to discretized problem depicted on Figure 2.4 p.41. In this simple case, the sub-domain Ω_2 has two disjoined interfaces, with Ω_1 on the top and Ω_3 on the bottom. For the sake of perspicuous demonstration we introduce here, the additional notation.

Notation 2.2 (Interface Manager for Inflated system)**.** *For* $1 \le I \ne J \le N$

- I_i *is a unique subset of* Ω_I *or* $\widetilde{\Omega}_I$ *which refers to interior nodes, i.e., nodes that do not lie on the interface and are not duplicated in any other sub-domain.*
- Γ_I *is set of indices which refer to nodes on the interface of* Ω_I.

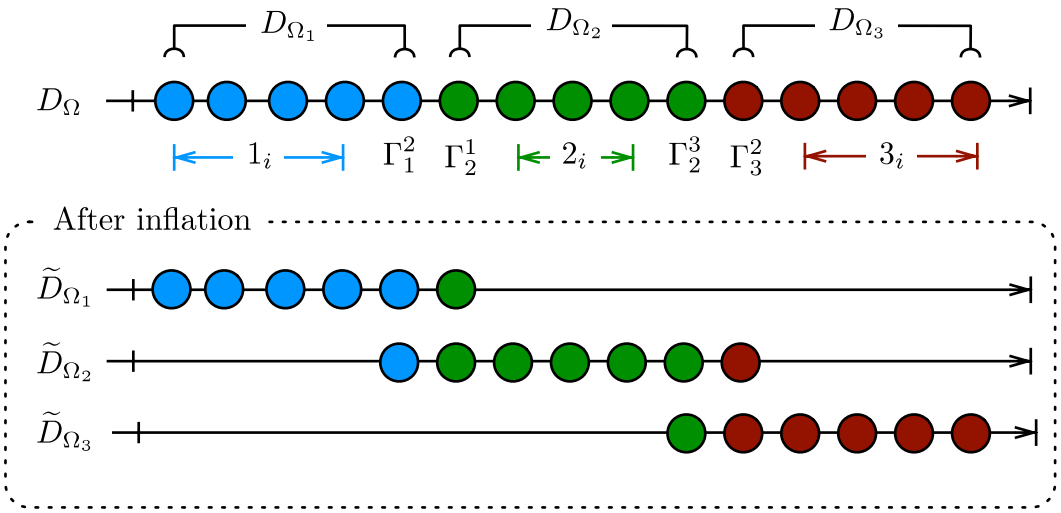


Figure 2.9: An 1D example of three subdomains inflation.

- *Let us take a partition of* Γ_I, $\Gamma_I = \cup_{J \neq I} \Gamma_I^J$ *such that* Γ_I^J *is a set of indices which refers to nodes ling on the interface between* Ω_I *and* Ω_J *only.*
- S_J^I *is the interface block operator to apply in* `Part` *with* `ID(I)`*, on its interface with* `Part` J.

$$\left[\begin{array}{cc|ccc|cc} A_{1_i 1_i} & A_{1_i \Gamma_1^2} & 0 & 0 & 0 & 0 & 0 \\ A_{\Gamma_1^2 1_i} & A_{\Gamma_1^2 \Gamma_1^2} & 0 & A_{\Gamma_1^2 \Gamma_2^1} & 0 & 0 & 0 \\ \hline 0 & 0 & A_{2_i 2_i} & A_{2_i \Gamma_2^1} & A_{2_i \Gamma_2^3} & 0 & 0 \\ 0 & A_{\Gamma_2^1 \Gamma_1^2} & A_{\Gamma_2^1 2_i} & A_{\Gamma_2^1 \Gamma_2^1} & 0 & 0 & 0 \\ 0 & 0 & A_{\Gamma_2^3 2_i} & 0 & A_{\Gamma_2^3 \Gamma_2^3} & 0 & A_{\Gamma_2^3 \Gamma_3^2} \\ \hline 0 & 0 & 0 & 0 & 0 & A_{3_i 3_i} & A_{3_i \Gamma_3^2} \\ 0 & 0 & 0 & 0 & A_{\Gamma_3^2 \Gamma_2^3} & A_{\Gamma_3^2 3_i} & A_{\Gamma_3^2 \Gamma_3^2} \end{array}\right] \left[\begin{array}{c} U_{1_i} \\ U_{\Gamma_1^2} \\ \hline U_{2_i} \\ U_{\Gamma_2^1} \\ U_{\Gamma_2^3} \\ \hline U_{3_i} \\ U_{\Gamma_3^2} \end{array}\right] = \left[\begin{array}{c} F_{1_i} \\ F_{\Gamma_1^2} \\ \hline F_{2_i} \\ F_{\Gamma_2^1} \\ F_{\Gamma_2^3} \\ \hline F_{3_i} \\ F_{\Gamma_3^2} \end{array}\right]. \tag{2.14}$$

When the system (2.14) is inflated, we can extend its operator by additional sub-operators

which acts on the interfaces. Modified in such a way operator, has a following form:

$$\left[\begin{array}{ccc|ccccc|ccc}
A_{1_i1_i} & A_{1_i\Gamma_1^2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
A_{\Gamma_1^2 1_i} & A_{\Gamma_1^2\Gamma_1^2} & A_{\Gamma_1^2\Gamma_2^1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & A_{\Gamma_2^1\Gamma_1^2} & A_{\Gamma_2^1\Gamma_2^1}+S_2^1 & A_{\Gamma_2^1 2_i} & -S_2^1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & A_{2_i2_i} & A_{2_i\Gamma_2^1} & A_{2_i\Gamma_2^3} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & A_{\Gamma_2^1 2_i} & A_{\Gamma_2^1\Gamma_2^1} & 0 & A_{\Gamma_2^1\Gamma_1^2} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & A_{\Gamma_2^3 2_i} & 0 & A_{\Gamma_2^3\Gamma_2^3} & 0 & A_{\Gamma_2^3\Gamma_3^2} & 0 & 0 & 0 \\
A_{\Gamma_1^2 1_i} & -S_1^2 & 0 & 0 & A_{\Gamma_1^2\Gamma_2^1} & 0 & A_{\Gamma_1^2\Gamma_1^2}+S_1^2 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & A_{\Gamma_3^2\Gamma_2^3} & 0 & A_{\Gamma_3^2\Gamma_3^2}+S_3^2 & A_{\Gamma_3^2 3_i} & -S_3^2 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & A_{3_i3_i} & A_{3_i\Gamma_3^2} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & A_{\Gamma_3^2 3_i} & A_{\Gamma_3^2\Gamma_3^2} & A_{\Gamma_3^2\Gamma_2^3} \\
0 & 0 & 0 & A_{\Gamma_2^3 2_i} & 0 & -S_2^3 & 0 & 0 & 0 & A_{\Gamma_2^3\Gamma_3^2} & A_{\Gamma_2^3\Gamma_2^3}+S_2^3
\end{array}\right] \tag{2.15}$$

2.5.3 Implementation

To enhance algebraically `Part` (subdomain) by new interface conditions. We need first to perform, at least one-level inflation, in order to build `Extra Indices Set`, i.e., include all current interface nodes to proceeded subdomain. In this way the `Interface Manager`; a special ADDMlib class for managing new interfaces, will modify only locally duplicated values of the original operator.

New interface conditions has a form of small, square block matrix, define for each `Part`. It can be arbitrary or automatically constructed (see Chapter 3 p.66). It size is determined by `Extra Indices Set` i.e., a collection of indices gathered from interface through inflation process. For one-level inflation, the size of new interface block-matrix is exactly equal to `Extra Indices Set`. For deeper inflation (more then one level), the sub-blocks size is determined by the number of new extra indices, added during last inflation pass. Thus, columns indices of the interface operators agree with extra indices corresponding with nodes lying on the interface of subdomain.

When new interface operator for a given `Part` is defined, the `Interface Manager` adds it to corresponding endomorphic `Partial Operator` by modifying extra rows i.e., those rows which were duplicated during last pass of inflation algorithm. To keep algebraic equivalence with original operator, the opposite values must be placed in the original position of extra indices i.e., the opposite sub-operator need to be define in the original exomorphic `Partial Operator` for any interface sub-block define on extra indices in endomorphic `Partial Operator`. For better picture see Figure 2.10, where procedure of "extending" interface conditions is presented in form of additional step to inflation algorithm.

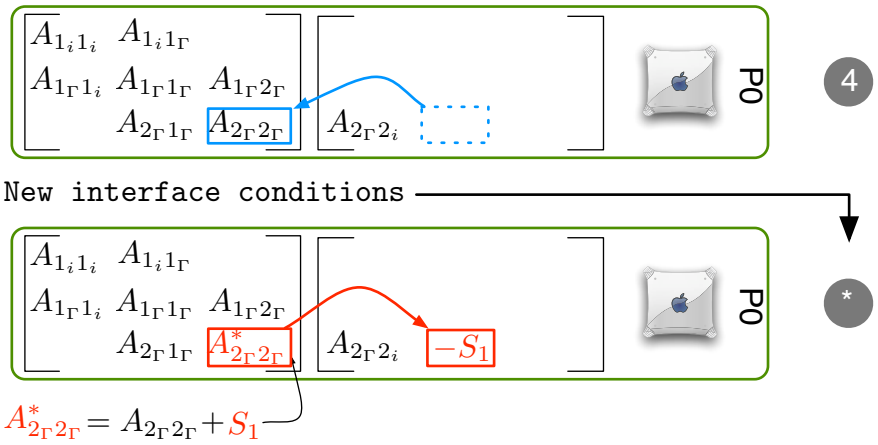


Figure 2.10: New interface condition as an additional step in inflation algorithm.

2.6 Sparse Patch Method

One possible way for enhancing interface conditions is to construct the *sparse patch* for each node $\mathcal{N}_i$ [12] lying on the interface Γ_I^J between subdomains D_{Ω_I} and D_{Ω_J} ($i \in \Gamma_I^J$, $I \neq J$), in order to obtain sparse approximation of the neighbour Schur complement. We modified the original algorithm (initially described in §1.3.2 on page 28) in order to fit it to the modified Schwarz method implemented in ADDMlib. Thus, in ADDMlib environment we use sparse patches to construct automatically an interface blocks S_i, defined in section 2.5.

Since `Interface Manager` requires at least one-level inflation, our *sparse patch* implementation is dedicated to overlapping sub-domains i.e., patches are build for nodes lying on the interfaces defined by inflation depth where indices originally did not belong to the sub-domain (`Part`).

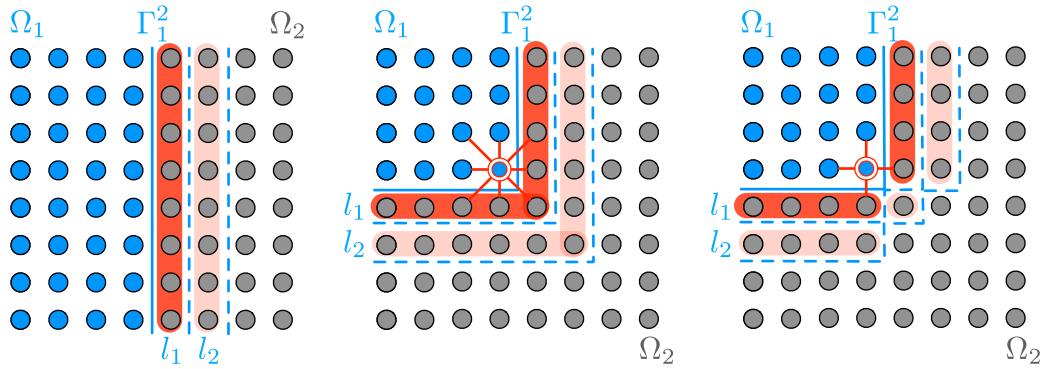
Patch parameters

Patch for node $\mathcal{N}_i$ on the interface Γ_I^J is defined by the set $\mathscr{P}_i^{\Gamma_I^J}$, in such a way that $\mathscr{P}_i^{\Gamma_I^J}$ is the subset of the neighboring subdomain D_{Ω_J} consists of indices which refer to joined together nodes $\mathcal{N}_j$ ($j \in D_{\Omega_J}$), in which at least one of them is connected with node $\mathcal{N}_i$ i.e., $a_{(i,j)} \neq 0$. Where $a_{(i,j)}$ is an element of underlying matrix A, and value $|a_{(i,j)}|$ specifies connection strength between those two nodes.

Each patch has an abstract geometry specified by two parameters; “width” and “depth”. The name of those two variables originate from graphical representation of the patch on two dimensional mesh, but even in more general cases (three dimensional meshes for instance) description of the patch geometry is still described by those two variables.

Patch width in ADDMlib’s implementation is the range of connection between the node and its neighboring nodes on the interface i.e., “width” equal one means: “include to patch computation all indices of nodes on the interface which are directly connected to current

12. See remark 2.3 at the end of following subsection.



(a) Fronts for simple decomposition (b) Fronts for nine point stencil numerical schema (c) Fronts for five point stencil numerical schema

Figure 2.11: The example of different "shape" of fronts l_d depending on the numerical scheme used for the discretization.

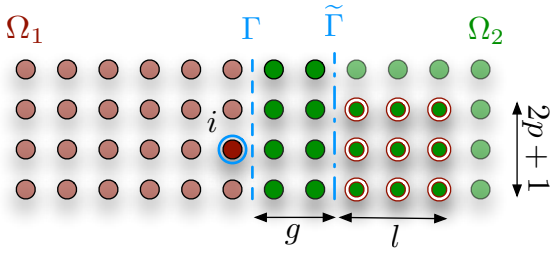
node". If the width is equal to two, we need to include to this set also indices of direct neighbours of those nodes and so on.

Direct neighbours of the processed node is an important issue for any algebraic method described in this report. This information is extracted from the matrix. Thus even for an arbitrary mesh or grid, the number of "direct neighbours" for each node (in algebraic sense) can vary as varies the numerical scheme used for the discretization.

The depth of the patch also refers to range of connections between processed node and its direct neighbours, but in this instance we look only for nodes which belong to neighbour subdomains. All nodes in neighbouring subdomain which are direct neighbours to nodes lying on the interface are called "first front" (l_1). If we go deeper i.e., we denote direct neighbours of nodes in first front and then neighbours of their neighbours, we can define another two fronts and so on.

All patch parameters can be easily illustrated with the help of the figure depicted below, and description of the actual parameters given as input to `Sparse Patch` subroutine, implemented in ADDMlib.

- g - **depth of inflation**. The number of nodes-fronts included to a sub-domain during inflation. The last included front, becomes new interface on which we build patches. The minimal value of g is 1 (see remark 2.2).
- p - **patch width**.
- l_{max} - **depth**. Maximum number of fronts in which we look for nodes to construct patches.



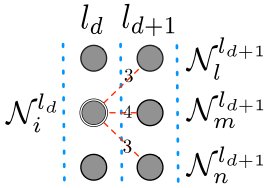
$$\texttt{Patch}(g, p, l_{max}, \alpha) \tag{2.16}$$

- α - **connectivity strategy**. Parameter used for steering the connectivity strategy (see description below).

Patch connectivity strategy

In previous subparagraph we showed that patch "depth" is expressed in terms of fronts, whose definition is closely connected with adjacency graph of the underlying matrix. In order to construct the set $\mathscr{P}_i^{\Gamma_I^J}$ we analyse connectivity between fronts starting from nodes on the interface (which we can consider as the front l_0), to nodes in front l_{max}.

In the general case, a chosen node $\mathscr{N}_n^d$ in front l_d can be connected with a number of nodes $\mathscr{N}_m^{d+1}$ in neighboring front l_{d+1}. The way how we choose which nodes include to the patch geometry, is called the "patch connectivity strategy". We control it by the float parameter α in following way:



- First we define a maximum connectivity strength $c_{max}(n)$ between $\mathscr{N}_n^d$ and $\mathscr{N}_m^{d+1}$.

$$c_{max}(n) = \max\left(|a_{(n,m)}| \,\middle|\, n : \mathscr{N}_n \in l_d, m : \mathscr{N}_m \in l_{d+1}, 0 \le d \le l_{max}\right)$$

- When c_{max} is computed we can decide (by fronts) which nodes to include to the patch geometry

$$\mathscr{P}_i^{\Gamma_I^J} = \cup_{d=0}^{l_{max}-1} \left\{ m : \mathscr{N}_m^{d+1} \in l_{d+1} \middle| a_{(n,m)} \neq 0 \wedge |a_{(n,m)}| \ge c_{max}(n) \times \alpha, n : \mathscr{N}_n \in l_d \right\}$$

From above is clear that for α = 0.0 we include to the patch geometry all nodes from l_{max} fronts in neighbouring subdomain, which are directly or indirectly (i.e., through their neighbours) connected to chosen (by patch "width") nodes on the interface Γ. In case of α = 1.0 this set is limited to only strongest connected nodes. The example is depicted on Figure 2.12.

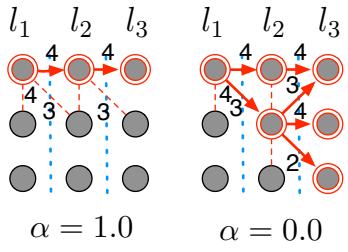


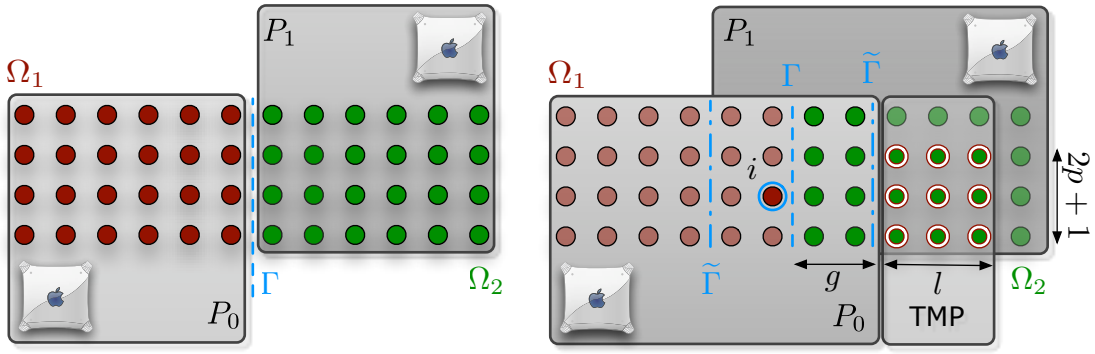
Figure 2.12: Connectivity strategy

Remark 2.2 (Depth of inflation in Sparse Patch routine)**.** *Inflation depth parameter g, is a strictly technical add-on to patch method described in §1.3.2. Its value describes after how many inflations repetition*

we will build patches. We group this parameter with variables denoting patch geometry in order to simplify the description of the numerical experiments which involves patch method with variable depths of inflation.

Remark 2.3 (Abstract definition of nodes)**.** *Whenever we speak about discrete constituents of the domain on which we perform computation, we name them "nodes". However, since the only source of information we have is the linear system which originates from discretization of the problem to solve. The definition of the nodes is very abstract and refers to "elements" used for discretization e.g., nodes of the grid in the finite-difference method, polygons of the mesh used in FEM methods and small volumes surrounding node point on a mesh in Finite volume methods. When we depict some property of the system or techniques used in our methods, the resulting figures refer to system discretized by finite-different method.*

2.6.1 Parallel implementation



(a) Initial data distribution.

(b) Data distribution during Sparse Patch computation.

Figure 2.13: Sub-domain data distribution for ADDMlib Sparse Patch algorithm.

The main difficulty in parallel, sparse patch computing is to localize the values of the global matrix, necessary for constructing new interface operator. For a given `Part`, patches are built over values extracted from operators associated with neighboring `Parts`. Thus, in general case, we have no local access to this data. However, we can avoid frequent message passing (for each "missing" value) by performing inflation. Hence, to increase total performance of Sparse Patch computation, we carry out $(g + l_{max})$-level inflation, in order to make local copy of data which will be used in patch computation.

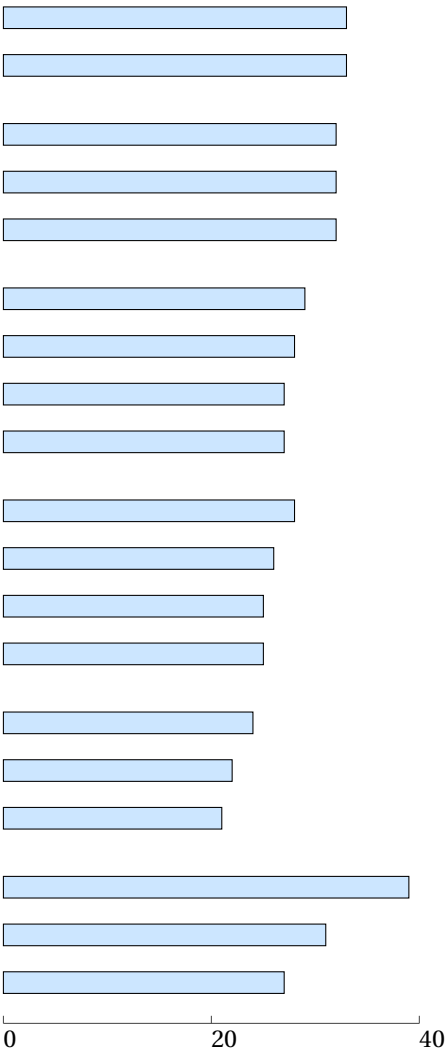
Illustration of data distribution related with this technique is depicted on Figure 2.13, where temporary data for `Part`(1) is denoted by TMP.

2.6.2 Numerical Experiments

In order to present influence of new interface conditions constructed via sparse patches, on roughly estimated condition number $\kappa_{\approx}$ and number of iterations, we proceeded experiment 2.1 p. 49 in which ASM algorithm was changed into MSM described in §2.5. All

the computations were performed for an arbitrary decomposition with $n_x = n_y = 50$ and $M_x = M_y = 5$. Variants of patch geometry are given according to arguments defined in (2.16). The results of the experiment are depicted below.

Method	$\kappa_{\approx}$	*n-iter*
MSM + Patch(1,1,1,1)	142.84	33
MSM + Patch(1,1,1,0)	137.84	33
MSM + Patch(1,2,1,1)	132.93	32
MSM + Patch(1,3,1,1)	130.27	32
MSM + Patch(1,3,1,0)	130.06	32
MSM + Patch(1,2,2,1)	105.36	29
MSM + Patch(1,2,3,1)	95.23	28
MSM + Patch(1,2,4,1)	91.27	27
MSM + Patch(1,2,4,0)	86.96	27
MSM + Patch(1,3,2,1)	97.55	28
MSM + Patch(1,3,3,1)	79.27	26
MSM + Patch(1,3,4,1)	76.05	25
MSM + Patch(1,3,4,0)	69.87	25
MSM + Patch(2,2,3,1)	67.86	24
MSM + Patch(3,2,3,0)	56.77	22
MSM + Patch(3,2,3,0)	55.57	21
ASM + 1 level Inflation	214.98	39
ASM + 2 level inflation	127.12	31
ASM + 3 level inflation	89.39	27



Influence of the different patch geometry on approximated condition number ($\kappa_{\approx}$) *and number of iterations (n-iter) in MSM method. Patch variants according to* (2.16)

Comments 2.6.1. *The effect of the patch interface condition is minimal for our tested cases. Other examples of the influence of the patch method are given in numerical result of depth 4. We see there that in connection with coarse grid correction, they improve convergence for Poisson like problem.*

Chapter 3

Enhanced Diagonal Optimal Interface Conditions

In this chapter I would like to define a new way for the algebraic approximation of optimal interface condition along with algorithm and data structures designed for parallel computing.

3.1 Sparse approximation of optimal conditions

For the sake of simplicity in this section we have only two subdomains ($D_\Omega = D_{\Omega_1} \cup D_{\Omega_2}$) and we focus on domain Ω_1 which we simply denote by 1 and its inflated counterpart by $\widetilde{1}$ (see Figure 3.1). Thus the optimal interface condition reads:

$$S^{opt}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1} := -A_{\widetilde{\Gamma}_1\widetilde{1}_c} A^{-1}_{\widetilde{1}_c\widetilde{1}_c} A_{\widetilde{1}_c\widetilde{\Gamma}_1}, \tag{3.1}$$

where $\widetilde{1}_c = \widetilde{\Omega} \setminus \widetilde{\Omega}_1$. Matrix $A^{-1}_{\widetilde{1}_c\widetilde{1}_c}$ is not readily available because it is distributed among processes. However not all entries of $A^{-1}_{\widetilde{1}_c\widetilde{1}_c}$ are used in (3.1), but only those whose columns are non zero rows of $A_{\widetilde{1}_c\widetilde{\Gamma}_1}$ and whose rows are non zero columns of $A_{\widetilde{\Gamma}_1\widetilde{1}_c}$. Therefore let us denote by $\widetilde{\Gamma}_1$ the boundary of $\widetilde{1}$ and lets assume that the graph of the matrix A is symmetric,

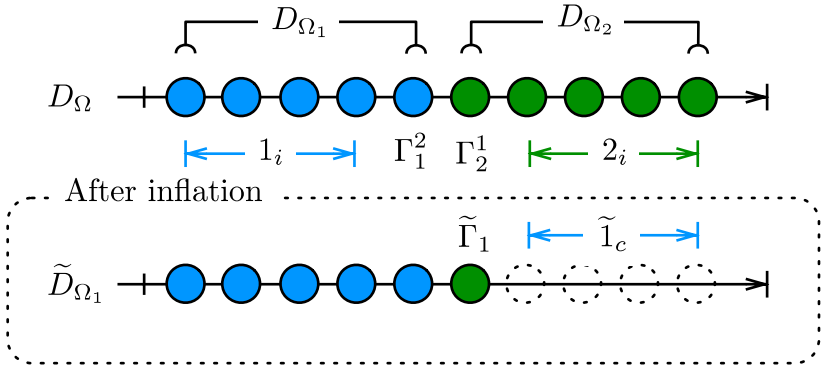


Figure 3.1: An 1D example of one level inflation for two subdomains.

then in (3.1) we only use

$$\left[A^{-1}_{\widetilde{1}_c\widetilde{1}_c}\right]_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1}.$$

Our goal is to approximate it by a sparse matrix keeping some filtering properties. More precisely, let V be a vector harmonic in $\widetilde{1}_c$, i.e.:

A harmonic vector property

$$A_{\widetilde{1}_c\widetilde{1}_c}V_{\widetilde{1}_c} + A_{\widetilde{1}_c\widetilde{\Gamma}_1}V_{\widetilde{\Gamma}_1} = 0 \tag{3.2}$$

Remark 3.1 (Decomposition used for harmonic vector V)**.** *Assuming the appropriate ordering of the unknowns and using straightforward notation for inflated system proposed in this section, the linear system* (2.8) *one can write in the following block form*

$$\begin{bmatrix} A_{1_i1_i} & A_{1_i\widetilde{\Gamma}_1} & 0 \\ A_{\widetilde{\Gamma}_11_i} & A_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1} & A_{\widetilde{\Gamma}_1\widetilde{1}_c} \\ 0 & A_{\widetilde{1}_c\widetilde{\Gamma}_1} & A_{\widetilde{1}_c\widetilde{1}_c} \end{bmatrix} \begin{bmatrix} U_{1_i} \\ U_{\widetilde{\Gamma}_1} \\ U_{\widetilde{1}_c} \end{bmatrix} = \begin{bmatrix} F_{1_i} \\ F_{\widetilde{\Gamma}_1} \\ F_{\widetilde{1}_c} \end{bmatrix} \tag{3.3}$$

where we assume that $A_{\widetilde{1}_c1_i} \equiv 0$*, which is satisfied for symmetric graph since* $A_{1_i\widetilde{1}_c} \equiv 0$ *(all indices in* 1_i *denote interior nodes of subdomain which "interact" only with its interface).*

Since the calculation of optimal interface blocks of type (3.1) is too costly, we seek rather a approximation to $S^{opt}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1}$ in the form

The optimal interface conditions approximation

$$S^{\approx}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1} := -A_{\widetilde{\Gamma}_1\widetilde{1}_c}\beta_{\widetilde{1}_c\widetilde{1}_c}A_{\widetilde{1}_c\widetilde{\Gamma}_1} \tag{3.4}$$

such that

$$-A_{\widetilde{\Gamma}_1\widetilde{1}_c}\beta_{\widetilde{1}_c\widetilde{1}_c}A_{\widetilde{1}_c\widetilde{\Gamma}_1}V_{\widetilde{\Gamma}_1} = S^{opt}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1}V_{\widetilde{\Gamma}_1} \tag{3.5}$$

where $\beta_{\widetilde{1}_c\widetilde{1}_c}$ is a sparse matrix to be chosen. Thus the optimality condition is verified only on a vector $V_{\widetilde{\Gamma}_1}$ defined on $\widetilde{\Gamma}_1$. Taking into account equation (3.2), it amounts to

$$\begin{aligned} -A_{\widetilde{1}_c\widetilde{\Gamma}_1}V_{\widetilde{\Gamma}_1} &= A_{\widetilde{1}_c\widetilde{1}_c}V_{\widetilde{1}_c} \\ -A_{\widetilde{\Gamma}_1\widetilde{1}_c}A^{-1}_{\widetilde{1}_c\widetilde{1}_c}A_{\widetilde{1}_c\widetilde{\Gamma}_1}V_{\widetilde{\Gamma}_1} &= A_{\widetilde{\Gamma}_1\widetilde{1}_c}V_{\widetilde{1}_c}. \end{aligned} \quad \cdot A_{\widetilde{\Gamma}_1\widetilde{1}_c}A^{-1}_{\widetilde{1}_c\widetilde{1}_c}$$

Let us now define the entries of sparse matrix $\beta_{\widetilde{1}_c\widetilde{1}_c}$.

Definition 3.1. *If* V *is a harmonic vector in* $\widetilde{1}_c$*, we take* $\beta_{\widetilde{1}_c\widetilde{1}_c}$ *to be a diagonal matrix defined by*

$\beta_{\tilde{\Gamma}_c\tilde{\Gamma}_c}$ operator

$$\beta_{\tilde{\Gamma}_c\tilde{\Gamma}_c} := \operatorname{diag}\left(-V_{\tilde{\Gamma}_c}./A_{\tilde{\Gamma}_c\tilde{\Gamma}_1}V_{\tilde{\Gamma}_1}\right) \tag{3.6}$$

and $\beta_{\tilde{\Gamma}_c\tilde{\Gamma}_c} = 0$ otherwise.

Remark 3.2 ("./" - element wise division)**.** *The "./" operator denotes element-wise division which is an operation dividing each entry in vector v with its corresponding entry in vector w, under assumption that vectors u and w have the same length. Thus we have*

$$\begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} ./ \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} v_1/w_1 \\ \vdots \\ v_n/w_n \end{bmatrix},$$

where n is size of vectors.

Proposition 3.1. *Let $\beta_{\tilde{\Gamma}_c\tilde{\Gamma}_c}$ satisfy* (3.6) *and assume that $A_{\tilde{\Gamma}_c\tilde{\Gamma}_1}V_{\tilde{\Gamma}_1}$ has no zero component on $\tilde{\Gamma}_1$. Then, we have*

$$S^{opt}_{\tilde{\Gamma}_1\tilde{\Gamma}_1}V_{\tilde{\Gamma}_1} = -A_{\tilde{\Gamma}_1\tilde{\Gamma}_c}\beta_{\tilde{\Gamma}_c\tilde{\Gamma}_c}A_{\tilde{\Gamma}_c\tilde{\Gamma}_1}V_{\tilde{\Gamma}_1}$$

Proof.

$$-A_{\tilde{\Gamma}_1\tilde{\Gamma}_c}\beta_{\tilde{\Gamma}_c\tilde{\Gamma}_c}A_{\tilde{\Gamma}_c\tilde{\Gamma}_1}V_{\tilde{\Gamma}_1} = A_{\tilde{\Gamma}_1\tilde{\Gamma}_c}V_{\tilde{\Gamma}_c} = -A_{\tilde{\Gamma}_1\tilde{\Gamma}_c}A^{-1}_{\tilde{\Gamma}_c\tilde{\Gamma}_c}A_{\tilde{\Gamma}_c\tilde{\Gamma}_1}V_{\tilde{\Gamma}_1}$$

which is by definition $S^{opt}_{\tilde{\Gamma}_1\tilde{\Gamma}_1}V_{\tilde{\Gamma}_1}$. □

In order to improve the approximation $S^{opt}_{\tilde{\Gamma}_1\tilde{\Gamma}_1}$ by our sparse matrix we do the following. Let $\beta_{\tilde{\Gamma}_c\tilde{\Gamma}_c}$ be a symmetric sparse operator that satisfies

$$\beta_{\tilde{\Gamma}_c\tilde{\Gamma}_c}A_{\tilde{\Gamma}_c\tilde{\Gamma}_c}V_{\tilde{\Gamma}_c} = V_{\tilde{\Gamma}_c}, \tag{3.7}$$

or equivalently using (3.2)

$$-\beta_{\tilde{\Gamma}_c\tilde{\Gamma}_c}A_{\tilde{\Gamma}_c\tilde{\Gamma}_1}V_{\tilde{\Gamma}_1} = V_{\tilde{\Gamma}_c}. \tag{3.8}$$

The optimal interface condition $S^{opt}_{\tilde{\Gamma}_1\tilde{\Gamma}_1}$ is approximated by

Definition of $S^{edoic}_{\tilde{\Gamma}_1\tilde{\Gamma}_1}$ operator

$$S^{edoic}_{\tilde{\Gamma}_1\tilde{\Gamma}_1} := -A_{\tilde{\Gamma}_1\tilde{\Gamma}_c}\left(2\beta_{\tilde{\Gamma}_c\tilde{\Gamma}_c} - \beta_{\tilde{\Gamma}_c\tilde{\Gamma}_c}A_{\tilde{\Gamma}_c\tilde{\Gamma}_c}\beta_{\tilde{\Gamma}_c\tilde{\Gamma}_c}\right)A_{\tilde{\Gamma}_c\tilde{\Gamma}_1} \tag{3.9}$$

The idea of this improvement originates from the following calculations: $\|(\mathscr{B}\mathscr{A} - I)\| \le \epsilon < 1$ leads to $\|(\mathscr{B}\mathscr{A} - I)^2\| \le \epsilon^2 < \epsilon$. Then, remarking that $(\mathscr{B}\mathscr{A} - I)^2 = \mathscr{B}\mathscr{A}\mathscr{B}\mathscr{A} - 2\mathscr{B}\mathscr{A} + I = I - (2\mathscr{B} - \mathscr{B}\mathscr{A}\mathscr{B})\mathscr{A}$, one concludes that $\mathscr{C} = 2\mathscr{B} - \mathscr{B}\mathscr{A}\mathscr{B}$ is better approximation of $\mathscr{A}^{-1}$ than $\mathscr{B}$ since $\|\mathscr{C} - I\| \le \epsilon^2 < \epsilon$.

Since new approximation is created by applying some enhancement to diagonal approximation defined in (3.6), we call it *Enhanced Diagonal Optimized Interface Conditions* (EDOIC). This new defined operator has three very interesting theoretical properties in the SPD case. The first two being independent of the choice of the symmetric matrix $\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}$.

Lemma 3.1. *If the original matrix is SPD, then we have*

- $\mathrm{A}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1} + \mathrm{S}^{edoic}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1}$ *is symmetric as* $\mathrm{A}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1} + \mathrm{S}^{opt}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1}$ *is.*
- *for all vector* v, $(\mathrm{S}^{edoic}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1} v, v) \geq (\mathrm{S}^{opt}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1} v, v)$. *Thus,* $\mathrm{A}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1} + \mathrm{S}^{edoic}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1}$ *is SPD as* $\mathrm{A}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1} + \mathrm{S}^{opt}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1}$ *is.*
- $\mathrm{S}^{edoic}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1}\mathrm{V}_{\widetilde{\Gamma}_1} = \mathrm{S}^{opt}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1}\mathrm{V}_{\widetilde{\Gamma}_1}$

Proof. Since symmetry is obvious, we prove the first two properties:

$$\begin{aligned}
\mathrm{S}^{edoic}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1} - \mathrm{S}^{opt}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1} &= \mathrm{A}_{\widetilde{\Gamma}_1\widetilde{\mathrm{I}}_c}\left(\mathrm{A}^{-1}_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c} - 2\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c} + \beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\mathrm{A}_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\right)\mathrm{A}_{\widetilde{\mathrm{I}}_c\widetilde{\Gamma}_1} \\
&= \mathrm{A}_{\widetilde{\Gamma}_1\widetilde{\mathrm{I}}_c}\mathrm{A}^{-1/2}_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\left(\mathbb{I}_{\widetilde{\Gamma}_1} - 2\mathrm{A}^{1/2}_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\mathrm{A}^{1/2}_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c} + \left(\mathrm{A}^{1/2}_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\mathrm{A}^{1/2}_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\right)^2\right)\mathrm{A}^{-1/2}_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\mathrm{A}_{\widetilde{\mathrm{I}}_c\widetilde{\Gamma}_1} \\
&= \mathrm{A}_{\widetilde{\Gamma}_1\widetilde{\mathrm{I}}_c}\mathrm{A}^{-1/2}_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\left(\mathbb{I}_{\widetilde{\Gamma}_1} - \mathrm{A}^{1/2}_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\mathrm{A}^{1/2}_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\right)^2\mathrm{A}^{-1/2}_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\mathrm{A}_{\widetilde{\mathrm{I}}_c\widetilde{\Gamma}_1}
\end{aligned}$$

Let us prove the last filtering property, using the harmonicity of the vector $(\mathrm{U}_{\widetilde{\Gamma}_1}, \mathrm{U}_{\widetilde{\mathrm{I}}^c})^{\mathrm{T}}$:

$$\begin{aligned}
\mathrm{S}^{edoic}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1}\mathrm{V}_{\widetilde{\Gamma}_1} &= -\ \mathrm{A}_{\widetilde{\Gamma}_1\widetilde{\mathrm{I}}_c}\left(2\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c} - \beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\mathrm{A}_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\right)\mathrm{A}_{\widetilde{\mathrm{I}}_c\widetilde{\Gamma}_1}\mathrm{V}_{\widetilde{\Gamma}_1} \\
&= \quad \mathrm{A}_{\widetilde{\Gamma}_1\widetilde{\mathrm{I}}_c}\left(2\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c} - \beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\mathrm{A}_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\right)\mathrm{A}_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\mathrm{V}_{\widetilde{\mathrm{I}}_c} \\
&= \quad \mathrm{A}_{\Gamma_1\widetilde{\mathrm{I}}_c}\left(2\mathrm{V}_{\widetilde{\mathrm{I}}_c} - \beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\mathrm{A}_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\mathrm{V}_{\widetilde{\mathrm{I}}_c}\right) \\
&= \quad \mathrm{A}_{\widetilde{\Gamma}_1\widetilde{\mathrm{I}}_c}\mathrm{V}_{\widetilde{\mathrm{I}}_c} \\
&= -\ \mathrm{A}_{\widetilde{\Gamma}_1\widetilde{\mathrm{I}}_c}\mathrm{A}^{-1}_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\mathrm{A}_{\widetilde{\mathrm{I}}_c\widetilde{\Gamma}_1}\mathrm{V}_{\widetilde{\Gamma}_1} \qquad = \mathrm{S}^{opt}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1}\mathrm{V}_{\widetilde{\Gamma}_1}
\end{aligned}$$

□

3.1.1 General case for arbitrary domain decomposition

We now consider a general case of domain decomposition ($\mathrm{D}_\Omega = \cup_{\mathrm{I}=1}^{\mathrm{N}}\mathrm{D}_{\Omega_\mathrm{I}}$) and for the sake of simplicity, we still focus on inflated subdomain $\widetilde{\mathrm{D}}_{\Omega_1} = \widetilde{1}$. Notice that in the general case there is no optimal interface condition like (3.1), which yields a convergence in a finite number of steps, but we may ask if in the other subdomains, the approximate solutions match and satisfy the equations but do not have necessarily the correct values, we have convergence in domain $\widetilde{\Omega}_1$ at the next step. In other words, the interface condition does not delay convergence. Then, the definition of this "nearly" optimal interface condition is still given by (3.1). This interface condition was approximated by (3.4) or (3.9) in the two-subdomain case. In general case, these formulas still make sense but the choice of the matrix $\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}$ is more problematic. Indeed, it seems difficult to have harmonic functions in the neighbours of domain $\widetilde{\Omega}_1$ that match on duplicated points. The most important thing to satisfy is $\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}$ to be uniquely defined. For this purpose, we define V using only original nodes, and we use it in formula (3.6) in order to compute values of $\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}$.

Before introducing more general way for computing operator $\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}$ we define additional notation:

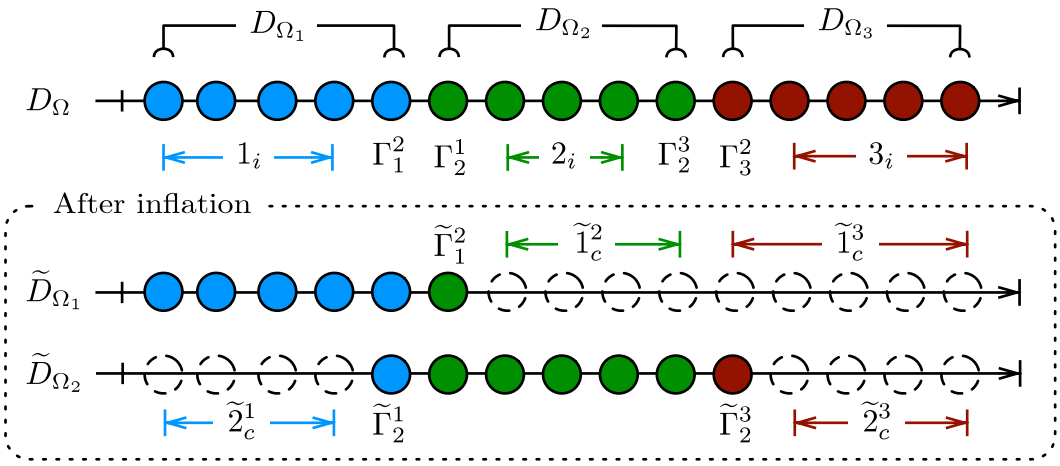


Figure 3.2: An 1D example of one and level inflation for three subdomains.

Notation 3.1 ($\widetilde{\mathrm{I}}_c$ decomposition)**.** *Let $\widetilde{\mathrm{I}}_c^{\mathrm{J}}$ is the set of indices associated with nodes which are not in the inflated part $\widetilde{\mathrm{I}}$ and belong to part* J *i.e.,*

- $\widetilde{\mathrm{I}}_c := \mathrm{D}_\Omega \setminus \widetilde{\mathrm{I}}$
- $\widetilde{\mathrm{I}}_c^{\mathrm{J}} \cap \widetilde{\mathrm{I}} = \emptyset$
- $\widetilde{\mathrm{I}}_c^{\mathrm{J}} := \widetilde{\mathrm{I}}_c \cap \mathrm{J}$
- $\widetilde{\mathrm{I}}_c = \cup_{\mathrm{J}=1}^{\mathrm{N}} \widetilde{\mathrm{I}}_c^{\mathrm{J}}$,

where N *is the number of subdomains.*

Notation 3.2 ($\widetilde{\Gamma}_{\mathrm{I}}$ decomposition)**.** *Let $\widetilde{\Gamma}_{\mathrm{I}}^{\mathrm{J}}$ be the set of indices associated with nodes which are in the inflated part $\widetilde{\mathrm{I}}$, they **lay** on its interface and originally belong to part* J *i.e.,*

- $\widetilde{\Gamma}_{\mathrm{I}}^{\mathrm{J}} := \widetilde{\Gamma}_{\mathrm{I}} \cup \mathrm{D}_{\Omega_{\mathrm{J}}}$,

where N *is the number of subdomains and $\widetilde{\Gamma}_{\mathrm{I}} \cup \mathrm{D}_{\Omega_{\mathrm{I}}} = \emptyset$.*

In order to perform the computation, we need to define a filling formula for operator $\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}$. Thus from (3.6) we can define

$$\left(\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}\right)_{ii} := -\left(\mathrm{V}_{\widetilde{\mathrm{I}}_c}\right)_i \cdot \Big/ \left(\mathrm{A}_{\widetilde{\mathrm{I}}_c\widetilde{\Gamma}_{\mathrm{I}}}\right)_{ij} \left(\mathrm{V}_{\widetilde{\Gamma}_{\mathrm{I}}}\right)_j, \tag{3.10}$$

where $(\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c})_{ii}$ is a diagonal entry of matrix $\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}$.

Taking into consideration the definition of new notations (3.2 & 3.1) we can observe that for an arbitrary decomposition, set $\widetilde{\mathrm{I}}_c$ can be divided into subsets related to different subdomains. In consequence, we can divide vectors $\mathrm{V}_{\widetilde{\mathrm{I}}_c}$ into sub-vector and operator $\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}$ into blocks. We choose therefore, to adapt (3.10) in the following manner:

Definition 3.2 (Arbitrary $\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}$)**.** *For each* $\mathrm{J} \neq \mathrm{I}$*, let* V_{J} *be a harmonic vector in subdomain* $\mathrm{D}_{\widetilde{\Omega}_{\mathrm{J}}}$ *which we can decompose in the following way:*

$$\mathrm{V}_{\mathrm{J}} = \begin{bmatrix} \mathrm{V}_{\widetilde{\Gamma}_{\mathrm{I}}^{\mathrm{J}}} \\ \mathrm{V}_{\widetilde{\mathrm{I}}_c^{\mathrm{J}}} \end{bmatrix}. \tag{3.11}$$

We compute for all $i \in \widetilde{\mathrm{I}}_c^{\mathrm{J}}$ and $j \in \widetilde{\Gamma}_{\mathrm{I}}^{\mathrm{J}}$:

Arbitrary $\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}$

$$\left(\beta_{\widetilde{\mathrm{I}}_c^{\mathrm{J}}\widetilde{\mathrm{I}}_c^{\mathrm{J}}}\right)_{ii} := -\left(\mathrm{V}_{\widetilde{\mathrm{I}}_c^{\mathrm{J}}}\right)_i \cdot \Big/ \left(\mathrm{A}_{\widetilde{\mathrm{I}}_c^{\mathrm{J}}\widetilde{\Gamma}_1^{\mathrm{J}}}\right)_{ij}\left(\mathrm{V}_{\widetilde{\Gamma}_1^{\mathrm{J}}}\right)_j \tag{3.12}$$

where it makes sense and zero otherwise.

3.1.2 Second order $\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}$ operator

From the formula (3.12) we know how to build and fill a sparse symmetric matrix $\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}$ out of local contributions $\beta_{\widetilde{\mathrm{I}}_c^{\mathrm{J}}\widetilde{\mathrm{I}}_c^{\mathrm{J}}}$ for $\mathrm{J} \neq 1$. In other words, for any part J, we shall build $\beta_{\widetilde{\mathrm{I}}_c^{\mathrm{J}}\widetilde{\mathrm{I}}_c^{\mathrm{J}}}$ such that

$$-\beta_{\widetilde{\mathrm{I}}_c^{\mathrm{J}}\widetilde{\mathrm{I}}_c^{\mathrm{J}}}\mathrm{A}_{\widetilde{\mathrm{I}}_c^{\mathrm{J}}\Gamma_1^{\mathrm{J}}}\mathrm{V}_{\Gamma_1^{\mathrm{J}}} = \mathrm{V}_{\widetilde{\mathrm{I}}_c^{\mathrm{J}}}. \tag{3.13}$$

However, vector $(\mathrm{A}_{\widetilde{\mathrm{I}}_c^{\mathrm{J}}\Gamma_1^{\mathrm{J}}}\mathrm{V}_{\Gamma_1^{\mathrm{J}}})$ is a sparse vector, while $\mathrm{V}_{\widetilde{\mathrm{I}}_c^{\mathrm{J}}}$ should be a full vector. Therefore block operator $\beta_{\widetilde{\mathrm{I}}_c^{\mathrm{J}}\widetilde{\mathrm{I}}_c^{\mathrm{J}}}$ cannot be a fully diagonal matrix and another way for filling must be developed.

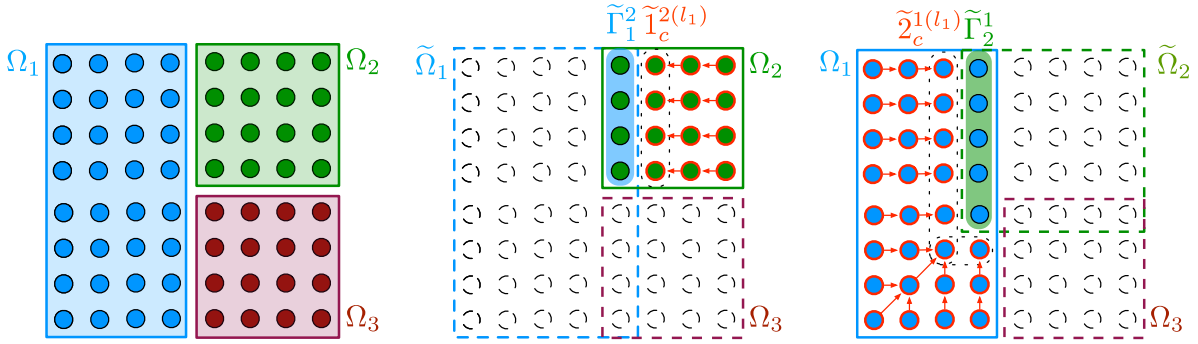
Firstly we can define for which indices in $\widetilde{\mathrm{I}}_c^{\mathrm{J}}$, the corresponding values in the sparse vector $(\mathrm{A}_{\widetilde{\mathrm{I}}_c^{\mathrm{J}}\Gamma_1^{\mathrm{J}}}\mathrm{V}_{\Gamma_1^{\mathrm{J}}})$ will be non-zero i.e., we can define a following set

$$\widetilde{\mathrm{I}}_c^{\mathrm{J}*} \quad := \quad \left\{ \quad i \in \widetilde{\mathrm{I}}_c^{\mathrm{J}} \quad \middle| \quad (\mathrm{A}_{\widetilde{\mathrm{I}}_c^{\mathrm{J}}\widetilde{\Gamma}_1^{\mathrm{J}}})_{ij}(\mathrm{V}_{\widetilde{\Gamma}_1^{\mathrm{J}}})_j \neq 0 \quad \right\}. \tag{3.14}$$

The important observation is that for all indices in $\widetilde{\mathrm{I}}_c^{\mathrm{J}*}$, the corresponding nodes are direct neighbours to nodes on the interface $\widetilde{\Gamma}_1^{\mathrm{J}}$. We can use this "geometrical" information to define a function

$$f : \widetilde{\mathrm{I}}_c^{\mathrm{J}} \to \widetilde{\mathrm{I}}_c^{\mathrm{J}*}, \tag{3.15}$$

in such a way, that f maps index of each node $\mathcal{N}_i$ ($i \in \widetilde{\mathrm{I}}_c^{\mathrm{J}}$) with index of the closest node $\mathcal{N}_j$ ($j \in \widetilde{\mathrm{I}}_c^{\mathrm{J}*}$). In order to determine a distance between two nodes, we use adjacency graph of the underlying matrix and we solve shortest path problem [4]. When mapping f is defined we use algorithm 5 to fill local operator $\beta_{\widetilde{\mathrm{I}}_c^{\mathrm{J}}\widetilde{\mathrm{I}}_c^{\mathrm{J}}}$. The example of shortest paths between nodes, is depicted on Figure 3.3.



(a) Square domain divided into three subdomains.

(b) Strongest connection within $\widetilde{1}_c^2$ set.

(c) Strongest connection within $\widetilde{2}_c^1$ set.

Figure 3.3: Examples of strongest connection "paths" between nodes which indices belong to complement sets.

Algorithm 5 Filling of the operator $\beta_{\widetilde{1}_c^J \widetilde{1}_c^J}$ using (3.12).

for $J \neq 1$ **do**
  for $i \in \widetilde{1}_c^J$ **do**
    temporary_value $:= -\left(V_{\widetilde{1}_c^J}\right)_i \Big/ \left(A_{\widetilde{1}_c^J \widetilde{\Gamma}_1^J}\right)_{f(i)j} \left(V_{\widetilde{\Gamma}_1^J}\right)_j$
    $\left(\beta_{\widetilde{1}_c^J \widetilde{1}_c^J}\right)_{i f(i)}$:= temporary_value
    if $i \neq f(i)$ **then**
      $\left(\beta_{\widetilde{1}_c^J \widetilde{1}_c^J}\right)_{f(i) i}$:= temporary_value
    end if
  end for
end for

3.2 Retrieving harmonic vector from solving system

In §3.1 we say that instead of computing the optimal blocks in form (3.1), we are interested in an approximation of the form (3.5).

Notice that due to the block preconditioning (the Schwarz method), the residuals ($r_i = \widetilde{F} - \widetilde{M}^{-1}\widetilde{A}\widetilde{U}_i$) are zero for the internal nodes in the subdomain. Thus, the vectors in the Krylov space $\mathcal{K}_m(\widetilde{M}^{-1}\widetilde{A}, r_0)$ are sub-domain wise harmonic.

If we had used another preconditioner, then for the small eigenvalue of the Ritz eigenvector are almost harmonic in the subdomain. Therefore, let us show now that carefully chosen eigen pair $(\lambda, \mathcal{V})$ of the inflated system (3.3) holds this assertion, thus it can be used to compute a sparse approximation of the optimal conditions (3.1).

Let $(\lambda, \mathcal{V})$ be any eigenpair of inflated system (3.3). $\mathcal{V}$ to agree with inflated operator $\widetilde{A}$ is decomposed in the following way

$$\mathcal{V} = \begin{bmatrix} \mathcal{V}_{1_i} \\ \mathcal{V}_{\widetilde{\Gamma}_1} \\ \mathcal{V}_{\widetilde{1}_c} \end{bmatrix} \quad \text{where} \quad \widetilde{A} = \begin{bmatrix} \begin{pmatrix} A_{1_i 1_i} & A_{1_i \widetilde{\Gamma}_1} \\ A_{\widetilde{\Gamma}_1 1_i} & A_{\widetilde{\Gamma}_1 \widetilde{\Gamma}_1} \end{pmatrix} & \begin{pmatrix} 0 \\ A_{\widetilde{\Gamma}_1 \widetilde{1}_c} \end{pmatrix} \\ \begin{pmatrix} 0 & A_{\widetilde{1}_c \widetilde{\Gamma}_1} \end{pmatrix} & \begin{pmatrix} A_{\widetilde{1}_c \widetilde{1}_c} \end{pmatrix} \end{bmatrix} \quad \text{and} \quad M^{-1} = \begin{bmatrix} \begin{pmatrix} A_{1_i 1_i} & A_{1_i \widetilde{\Gamma}_1} \\ A_{\widetilde{\Gamma}_1 1_i} & A_{\widetilde{\Gamma}_1 \widetilde{\Gamma}_1} \end{pmatrix}^{-1} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\ \begin{pmatrix} 0 & 0 \end{pmatrix} & \begin{pmatrix} A_{\widetilde{1}_c \widetilde{1}_c} \end{pmatrix}^{-1} \end{bmatrix}$$

then for left preconditioner we have

$$M^{-1}\widetilde{A} = \begin{bmatrix} \mathbb{I} & \begin{pmatrix} A_{1_i 1_i} & A_{1_i \widetilde{\Gamma}_1} \\ A_{\widetilde{\Gamma}_1 1_i} & A_{\widetilde{\Gamma}_1 \widetilde{\Gamma}_1} \end{pmatrix}^{-1} \begin{pmatrix} 0 \\ A_{\Gamma_1 \widetilde{1}_c} \end{pmatrix} \\ \begin{pmatrix} A_{\widetilde{1}_c \widetilde{1}_c} \end{pmatrix}^{-1} \begin{pmatrix} 0 & A_{\widetilde{1}_c \widetilde{\Gamma}_1} \end{pmatrix} & \mathbb{I} \end{bmatrix}.$$

Hence from the definition of the eigenvector we can write

$$M^{-1}\widetilde{A}\mathcal{V} = \lambda\mathcal{V} \tag{3.16a}$$

$$\begin{bmatrix} A_{1_i 1_i} & A_{1_i \widetilde{\Gamma}_1} \\ A_{\widetilde{\Gamma}_1 1_i} & A_{\widetilde{\Gamma}_1 \widetilde{\Gamma}_1} \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ A_{\widetilde{\Gamma}_1 \widetilde{1}_c} \mathcal{V}_{\widetilde{1}_c} \end{bmatrix} = (\lambda - 1) \begin{bmatrix} \mathcal{V}_{1_i} \\ \mathcal{V}_{\widetilde{\Gamma}_1} \end{bmatrix}$$

$$\begin{bmatrix} A_{\widetilde{1}_c \widetilde{1}_c} \end{bmatrix}^{-1} \begin{bmatrix} A_{\widetilde{1}_c \widetilde{\Gamma}_1} \mathcal{V}_{\widetilde{\Gamma}_1} \end{bmatrix} = (\lambda - 1) \begin{bmatrix} \mathcal{V}_{\widetilde{1}_c} \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ A_{\widetilde{\Gamma}_1 \widetilde{1}_c} \mathcal{V}_{\widetilde{1}_c} \end{bmatrix} = (\lambda - 1) \begin{bmatrix} A_{1_i 1_i} & A_{1_i \widetilde{\Gamma}_1} \\ A_{\widetilde{\Gamma}_1 1_i} & A_{\widetilde{\Gamma}_1 \widetilde{\Gamma}_1} \end{bmatrix} \begin{bmatrix} \mathcal{V}_{1_i} \\ \mathcal{V}_{\widetilde{\Gamma}_1} \end{bmatrix} \tag{3.16b}$$

$$\begin{bmatrix} A_{\widetilde{1}_c \widetilde{\Gamma}_1} \mathcal{V}_{\widetilde{\Gamma}_1} \end{bmatrix} = (\lambda - 1) \begin{bmatrix} A_{\widetilde{1}_c \widetilde{1}_c} \end{bmatrix} \begin{bmatrix} \mathcal{V}_{\widetilde{1}_c} \end{bmatrix} \tag{3.16c}$$

Now we can expand (3.16c)

$$A_{\widetilde{1}_c \widetilde{\Gamma}_1} \mathcal{V}_{\widetilde{\Gamma}_1} = \lambda \left(A_{\widetilde{1}_c \widetilde{1}_c} \mathcal{V}_{\widetilde{1}_c} \right) - A_{\widetilde{1}_c \widetilde{1}_c} \mathcal{V}_{\widetilde{1}_c},$$

thus if $\lambda \simeq 0$ we get

$$A_{\widetilde{1}_c \widetilde{\Gamma}_1} \mathcal{V}_{\widetilde{\Gamma}_1} + A_{\widetilde{1}_c \widetilde{1}_c} \mathcal{V}_{\widetilde{1}_c} = 0 \tag{3.17}$$

As a result we could use values of eigenpair $(\lambda, \mathcal{V})$ (for sufficiently small λ) in order to build operator $\beta_{\widetilde{1}_c \widetilde{1}_c}$ (3.6). Thus the construction of diagonal approximation of optimal interface condition can be achieved by looking for eigenvectors with extreme eigenvalues in block-preconditioned matrix $\widetilde{A}$.

Right preconditioned system

For the right-preconditioned system $\widetilde{A}M^{-1}U^\star = F$, with $U = M^{-1}U^\star$, let first prove a general result :

Proposition 3.2. *If* $(\lambda, \mathcal{V}_L)$ *is an eigenpair of* $\left(M^{-1}A\right)$ *then* $(\lambda, \mathcal{V}_R = M\mathcal{V}_L)$ *is an eigenpair of* $\left(AM^{-1}\right)$.

Proof.

$$\begin{aligned} M^{-1}Ax &= \lambda x \\ Ax &= \lambda Mx \qquad \cdot M \\ \left(AM^{-1}\right)Mx &= \lambda Mx. \end{aligned}$$

□

It yields that in the right-preconditioned system, after the eigenvector $\mathcal{V}_R$ of AM^{-1} is calculated, one must compute $\mathcal{V} = M^{-1}\mathcal{V}_R$. It is not an eigenvector for A but for $M^{-1}A$, and thus we can construct operator $\beta_{\widetilde{1}_c\widetilde{1}_c}$ (3.6) using filtering property (3.17).

Retrieving approximate eigenvector from GMRES solver

Many iterative methods for the solution of linear system and the computation of (selected) eigenvalues make use of Krylov subspaces. Thus, for a given real, nonsingular matrix $A \in \mathbb{R}^{n\times n}$ and a vector $r_0 \in \mathbb{R}^n$, the Krylov subspaces

$$\mathcal{K}_m(A, r_0) = \text{SPAN}\left\{r_0, Ar_0, A^2 r_0, \ldots, A^{m-1} r_0\right\} \tag{3.18}$$

for $m = 1, 2, \ldots, n$ form a nested sequence of subspaces.

The computational kernel of GMRES [52] is the Arnoldi process which computes the orthonormal basis W_m for the Krylov subspace $\mathcal{K}_m(A, r_0)$. Since the Arnoldi basis is orthonormal, $W_m = (w_1\ w_2\ \ldots\ w_m)$ is an orthogonal matrix ($W_m \in \mathbb{R}^{n\times m}$). In the orthogonalisation process the scalars h_{ij} are computed so that the square upper Hessenberg matrix $H_m \in \mathbb{R}^{m\times m}$ satisfies the fundamental relation

$$AW_m = W_m H_m + h_{m+1,m} w_{m+1} e_m^H = W_{m+1}\overline{H}_m. \tag{3.19}$$

The rectangular upper Hessenberg matrix $\overline{H}_m \in \mathbb{R}^{(m+1)\times m}$ is the square upper Hessenberg matrix H_m supplemented with an extra row $(0\ \ldots\ 0\ h_{m+1,m})$. From (3.19) we can derive the following expression for H_m:

$$H_m = W_m^H A W_m. \tag{3.20}$$

The eigenvalues of H_m are called Ritz values and they approximate the eigenvalues of A. Thus, in practice, the best way to approximate eigenvector $\mathcal{V}$ of A is to compute the Ritz pair (z_m, λ), where z_m are the eigenvectors of matrix H_m extracted from GMRES solver [29]. If z_m is an eigenvector for H_m, then $\mathcal{V}_m = W_m z_m$ is almost an eigenvector of A, for the same eigenvalue i.e.,

$$\begin{aligned} A\mathcal{V}_m &\simeq W_m H_m W_m^H W_m z_m &&= \\ &= W_m H_m z_m &&= \\ &= W_m \lambda z_m &&= \lambda \mathcal{V}_m. \end{aligned} \tag{3.21}$$

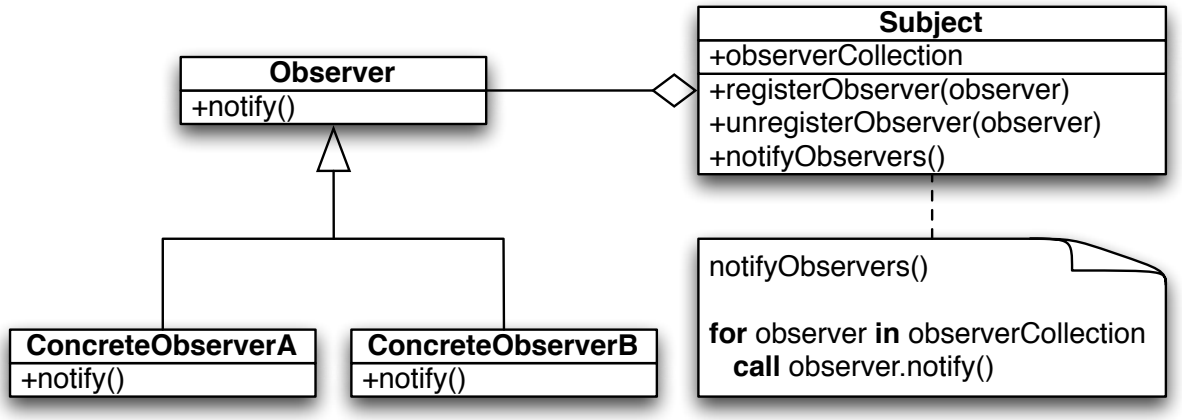


Figure 3.4: UML diagram of Observer pattern.

3.3 Parallel implementation

The computing process of new interface condition in form (3.9) is divided into two stages. During the first phase we construct operators $\beta_{\widetilde{\Gamma}_c\widetilde{\Gamma}_c}$ in close collaboration with krylov solver. Next, the more demanding computation is performed - the construction of interface operators $S^{edoic}_{\widetilde{\Gamma}_I\widetilde{\Gamma}_I}$. We subdivided this section according to this splitting.

3.3.1 Implementation of β operators

Implementation of routine which compute β operators, is a good example of the Observer pattern[25]i.e., a software design pattern in which an object, called the subject, maintains a list of its dependants, called observers, and notifies them automatically of any state changes, usually by calling one of their methods (see UML [1] diagram depicted on figure 3.4).

In ADDMlib the Krylov solver objects (GMRES/FGMRES) are the "subjects" which notify all their "observers" about progress in computation i.e., about current number of iteration and current residual norm. Thus, an "observer" (in ADDMlib this is an object which computes and keeps sparse data of matrixes β) in answer can analyse this simple data and in chosen moment (after given number of iterations or when it will discover stagnation in solver iterations) can perform an additional computation like calculation of approximated eigenvector.

Approximated eigenvector

In order to compute approximated eigenpair $(\mathcal{V},\lambda)$, the β matrix object (notified by "subject" after m iterations), extracts from solver a square upper Hessenberg matrix $H_m \in \mathbb{R}^{m\times m}$ and orthonormal basis W_m for the Krylov subspace used by linear solver [2]. H_m is a small

1. **U**nified **M**odelling **L**anguage is a standardised general-purpose modelling language in the field of software engineering.

2. Usually in our experiments the Arnoldi loop constructs an orthogonal basis of the left-preconditioned Krylov subspace for inflated operator i.e., $\mathcal{K}_m(\widetilde{M}^{-1}\widetilde{A}, r_0)$

matrix which is local for each process, while W_m consists of m - `DDMVectors` (see §2.2.1).

After extraction, ADDMlib performs eigendecomposition [3] of H_m in order to obtain $H_m = ZDZ^{-1}$, where D is a diagonal matrix formed from the sorted by a magnitude eigenvalues of H_m, and the columns of Z are the corresponding eigenvectors of H_m ($z_1^* \ \dots \ z_m^*$).

Next we pick from D the smallest eigenvalue λ and its corresponding eigenvector z_m^* from Z, in order to perform the final computation:

$$\mathcal{V}_m = \left(w_1 \cdot z_m^*(1) + w_2 \cdot z_m^*(2) + \ldots + w_m \cdot z_m^*(m)\right). \tag{3.22}$$

Computing the approximated eigenvector $\mathcal{V}$ is "cheap" and does not involve any MPI communication because each process has its own, local copy of the matrix H_m. Thus, a *vector update* operations in (3.22) is performed on local `Partial Vectors` and duplicated on each process, the vector z_m^*.

Filling β operators

When approximated eigenvector is computed, we can use it as a harmonic vector V in order to fill operators β via formula (3.12). As in case of approximation of eigenvector $\mathcal{V}$, this is also a local operation since all the necessary data for the computation consists in endomorphic `Partial Operators` and corresponding `Partial Vectors` (local contributors of $\mathcal{V}$).

The important observation is that each `Part` needs to construct a number of β operators equal to number of their neighbours, which is a consequence of different definition of set $\widetilde{\Gamma}_c^J$ for varying part `ID`= I i.e., for fixed J and for all $L \neq M$, $\widetilde{L}_c^J \neq \widetilde{M}_c^J$. For the same reason we need to define unique map f (3.15) for each operator. But this operation is local and independent of vector V used in filling formula, thus we can perform it as a pre processing.

3.3.2 Computing $S^{edoic}_{\widetilde{\Gamma}_I\widetilde{\Gamma}_I}$

Because the *linear algebra kernel* implemented in ADDMlib has no general routines for sparse matrix multiplication. We need to perform all matrix-matrix multiplications in (3.9) "manually" i.e., we have to retrieve matrices $A_{\widetilde{\Gamma}_I\widetilde{I}_c}$, $A_{\widetilde{I}_c\widetilde{\Gamma}_I}$ and submatrices of $A_{\widetilde{I}_c\widetilde{I}_c}$ and $\beta_{\widetilde{I}_c\widetilde{I}_c}$ for all $1 \leq I \leq N$ and compute (3.9). However we can notice that we do not need to have access to all entries in $A_{\widetilde{I}_c\widetilde{I}_c}$, since matrix $A_{\widetilde{\Gamma}_I\widetilde{I}_c}$ is located where `Part` with `ID` I is, whereas the three other matrices are located where the neighbours of `Part` I are. Thus, it makes sense to drive the computation of $S^{edoic}_{\widetilde{\Gamma}_I\widetilde{\Gamma}_I}$ by the most right factor of $A_{\widetilde{I}_c\widetilde{\Gamma}_I}$ and split this computation in the same manner as we can split $\widetilde{\Gamma}_I$ ($\widetilde{\Gamma}_I = \cup_{J\neq I}\widetilde{\Gamma}_I^J$).

In contrary to the computation of the matrices $\beta_{\widetilde{I}_c\widetilde{\Gamma}_I}$, the construction of the new interface operators needs some data exchanges between `Parts`. Nevertheless we have managed to minimize number of messages exchange between the processes by dividing the formula (3.9)

3. Because of the small size of the matrix, this computation is performed by routines implemented in well known LAPACK library (Linear Algebra PACKage [5])

into four distinct and local operations separated by some *block data exchange*[4]. The whole process is depicted on Figure 3.5 and we dedicate to it a following description:

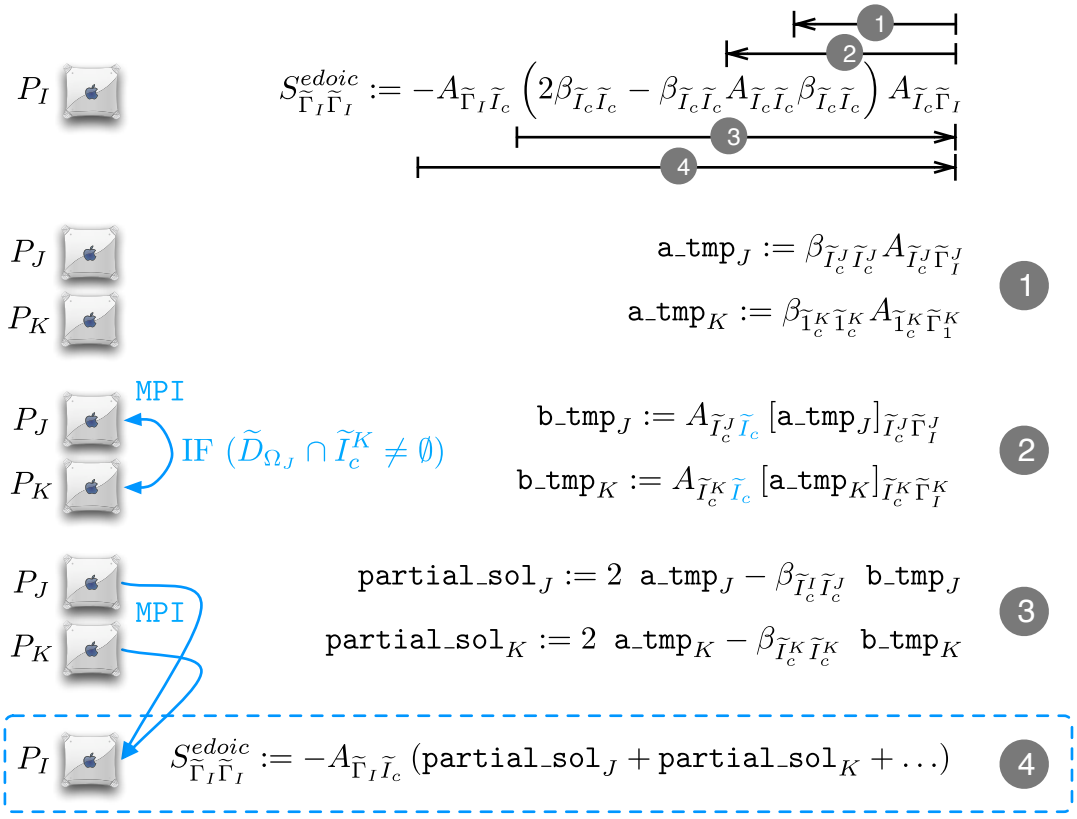


Figure 3.5: Four main steps of computation $S^{edoic}_{\widetilde{\Gamma}_1\widetilde{\Gamma}_1}$.

First step - $\mathtt{a\_tmp}_J$ Thanks to inflation process, indices in $\widetilde{\Gamma}_1^J$ are duplicated from `Part` with `ID` J and in order to be able to track this duplication, they are marked as *shared indices* (see §2.3). Thus, since indices in $\widetilde{I}_c^J$ are by definition part of D_{Ω_J}, all entries of operator $A_{\widetilde{I}_c^J\widetilde{\Gamma}_1^J}$ can be found in endomorphic `Partial Operator` of `Part` J. Therefore for all $j \in$ *shared indices* of the `Part` J we loop over the entries of the corresponding column. The row number of the entry is denoted by k and we have to decide if we keep it, i.e., if $k \in \widetilde{I}_c^J$. Each value collected in this way is multiply by non-zeros in column k of the matrix $\beta_{\widetilde{I}_c^J\widetilde{I}_c^J}$ and in consequence we get

$$\left(\mathtt{a\_tmp}_J\right)_{lj} = \sum_k \left(\beta_{\widetilde{I}_c^J\widetilde{I}_c^J}\right)_{lk}\left(A_{\widetilde{I}_c^J\widetilde{\Gamma}_1^J}\right)_{kj}. \tag{3.23}$$

We store this sparse matrix separately in order to reuse it in steep three.

Important observation is that, computation in the first step are local to a given process. It is due to fact that operator $\beta_{\widetilde{I}_c\widetilde{I}_c}$ is defined `Part` wise, thus rows indices in operator $A_{\widetilde{I}_c\widetilde{\Gamma}_I^J}$ are limited to set $\widetilde{I}_c^J$ which is a subset of D_{Ω_J}.

4. Instead of exchanging single values of matrix between processes (when its needed), we first collect all of them, in order to send one "big" message consisting data of sparse sub-matrix.

Second step - $\texttt{partial\_sol}_{\mathrm{J}}$ Matrix-matrix product witch involves operator $A_{\widetilde{\Gamma}_c\widetilde{\Gamma}_c}$ requires very careful handling, since during first step of inflation process, non-zero values from exomorphic `Partial Operators` with the same `PartOutID` are moved to corresponding endomorphic `Partial Operator`. In terms of the computation a formula (3.9) it means that in case when $\widetilde{\Gamma}_c^{\mathrm{J+K}} = \widetilde{\mathrm{D}}_{\Omega_{\mathrm{J}}} \cap \widetilde{\Gamma}_c^{\mathrm{K}} \neq \emptyset$, we can retrieve from endomorphic `Partial Operator` J a small sparse matrix $A_{\widetilde{\Gamma}_c^{\mathrm{J}}\widetilde{\Gamma}_c^{\mathrm{J+K}}}$. Which we need to send to `Part` K in order to obtain partial result: $\texttt{part\_tmp}_{\mathrm{J+K}} = A_{\widetilde{\Gamma}_c^{\mathrm{J}}\widetilde{\Gamma}_c^{\mathrm{J+K}}}[\texttt{tmp}_{\mathrm{K}}]_{\widetilde{\Gamma}_c^{\mathrm{J+K}}\widetilde{\Gamma}_c^{\mathrm{J+K}}}$. Next, the $\texttt{part\_tmp}_{\mathrm{J+K}}$ is send back to `Part` J in order to define

$$\texttt{b\_tmp}_{\mathrm{J}} = A_{\widetilde{\Gamma}_c^{\mathrm{J}}\widetilde{\Gamma}_c^{\mathrm{J}}}\texttt{a\_tmp}_{\mathrm{J}} + \texttt{part\_tmp}_{\mathrm{J+K}} = \sum_l \sum_k \left(A_{\widetilde{\Gamma}_c^{\mathrm{J}}\widetilde{\Gamma}_c}\right)_{ml} \left(\beta_{\widetilde{\Gamma}_c\widetilde{\Gamma}_c^{\mathrm{J}}}\right)_{lk} \left(A_{\widetilde{\Gamma}_c^{\mathrm{J}}\widetilde{\Gamma}_{\mathrm{I}}^{\mathrm{J}}}\right)_{kj} \tag{3.24}$$

Third steep - $\texttt{partial\_sol}_{\mathrm{J}}$ Starting from this steep, we drive computation of matrix products from the left side (by rows indices not columns). It is due to fact that all operators at this level are well described i.e., they have own containers and we do not need retrieve them from underling `Partial Operator`. Moreover, the computation is local to a given process and we reuse in it partial results from previous steps. Thus, we can easily compute

$$\left[\texttt{partial\_sol}_{\mathrm{J}}\right]_{\widetilde{\Gamma}_c^{\mathrm{J}}\widetilde{\Gamma}_{\mathrm{I}}^{\mathrm{J}}} = 2\ \texttt{a\_tmp}_{\mathrm{J}} - \beta_{\widetilde{\Gamma}_c^{\mathrm{J}}\widetilde{\Gamma}_c^{\mathrm{J}}}\ \texttt{b\_tmp}_{\mathrm{J}}, \tag{3.25}$$

and send it to `Part` I in order to finish computation.

Fourth step - finishing computation When `Part` with `ID` I will gather all $\texttt{partial\_sol}_{\mathrm{J}}$ from its neighbours, it can easily combine them into one operator

$$\left[\texttt{partial\_sol}_{\mathrm{J}} + \texttt{partial\_sol}_{\mathrm{K}} + \ldots\right]_{\widetilde{\Gamma}_c\widetilde{\Gamma}_{\mathrm{I}}} \tag{3.26}$$

and finish computation by multiply it by operator $A_{\widetilde{\Gamma}_{\mathrm{I}}\widetilde{\Gamma}_c}$.

Entries of the matrix $A_{\widetilde{\Gamma}_{\mathrm{I}}\widetilde{\Gamma}_c}$ are readily available in the duplicated lines of the exomorphic `Partial Operators` which `PartOutID` = I. Therefore, this is a local computation since after inflation all `Partial Operators` with the same `PartOutID` are stored on the same process (see §2.3 p. 45).

3.4 Numerical results

For all experiment in this section let us consider Laplace's equation $-\Delta(\mathbf{u}) = 0$, discretized using P1-type finite elements on 2D domain triangulated by the *Delaunay-Voronoi*-type algorithm. The right hand side of resulting linear system is a function f which gives random values from set $\langle 1,2)$ (fixed for all variants of test). For domain decomposition we use graph-partitioner.

We inflate once and solve the resulting discrete system using GMRES left preconditioned by ASM or MSM. The initial guess (in both cases) is chosen to be $\mathbf{u}^{(0)} = 0$ and the stopping criterion $\|r_i\| \leq tol \cdot \|r_0\|$ for $tol = 1 \times 10^{-6}$. The estimated condition number is given as $\kappa_{\approx} = \lambda_{max}/\lambda_{min}$ where $\lambda_{\{min,max\}}$ are the approximated, extreme eigenvalues of $(\mathrm{M}^{-1}\widetilde{\mathrm{A}})$.

In order to present how the new interface condition influences, the roughly estimated the roughly estimate condition number $\kappa_{\approx}$ and the number of iterations, the resulting linear system is solved twice. First solver is preconditioned by ASM and then second by MSM. Moreover, the first n iterations of ASM algorithm are used to find an approximated eigenvector $\mathcal{V}$. Next the vector $\mathcal{V}$ is used in construction of `Part` wise operators $\beta_{\widetilde{\mathrm{I}}_c\widetilde{\mathrm{I}}_c}$. After that, we are able to compute interface sub-operators $\mathrm{S}_{\widetilde{\Gamma}_\mathrm{I}\widetilde{\Gamma}_\mathrm{I}}$ (3.9) in order to enhance inflated operator $\widetilde{\mathrm{A}}$ and solve it by the MSM algorithm (§2.5).

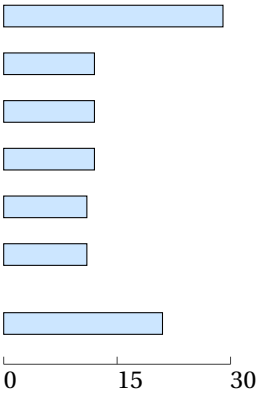
3.4.1 EDOIC and quality of eigenvector approximation

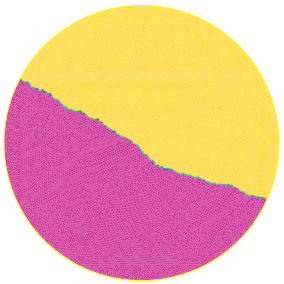
In the following experiment we test how the quality of eigenvector approximation influences the robustness of the EDOIC type interface conditions. The computational domain is chosen to be a triangulated (27604 triangles) unit disk. For decomposition we use SCOTCH partitioner over adjacency graph originate from underlying matrix.

EDOIC(n) denotes a variant of experiment in which the eigenvector has been approximated after n iterations of reference solver i.e., "(ref) ASM".

Two sub-domain case

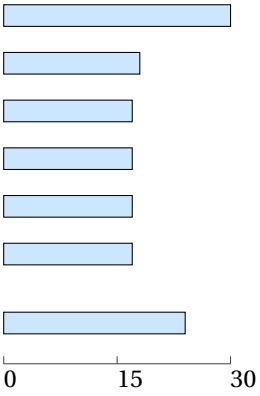
Method	$\kappa_{\approx}$	*n-iter*
MSM+EDOIC(1)	46.13	29
MSM+EDOIC(2)	5.72	12
MSM+EDOIC(3)	5.92	12
MSM+EDOIC(6)	5.78	12
MSM+EDOIC(10)	4.76	11
MSM+EDOIC(20)	5.16	11
(ref) ASM	27.03	21

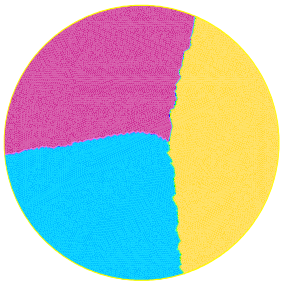




Three sub-domain case

Method	$\kappa_{\approx}$	*n-iter*
MSM+EDOIC(1)	64.13	32
MSM+EDOIC(2)	21.80	18
MSM+EDOIC(3)	20.35	17
MSM+EDOIC(6)	14.56	17
MSM+EDOIC(10)	14.37	17
MSM+EDOIC(20)	14.36	17
(ref) ASM	36.47	24



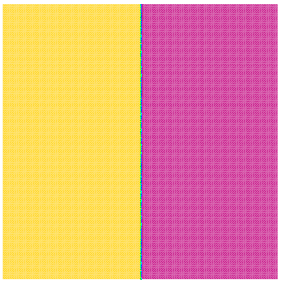


Comments 3.4.1. *We see that if the Ritz eigenvector is computed after at least two iterations, numerical results are improved by EDOIC*

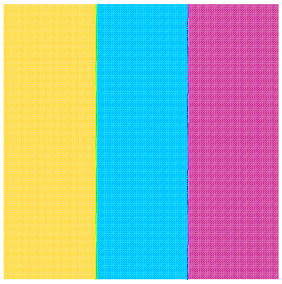
3.4.2 EDOIC *versus* number of subdomains

In the following experiments we present how the number and the shape of the subdomains influence on the robustness of the EDOIC-type interface conditions. The computational domains are chosen to be an unit square with fixed size (150×150 vertices) and a complex domain which is the union of a disc and rectangle. New interface conditions are tested with different partitions depicted along with the results.

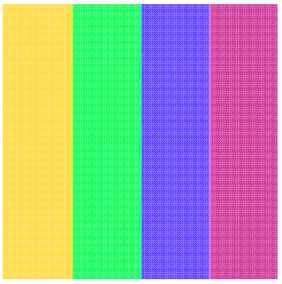
Experiment with unite square



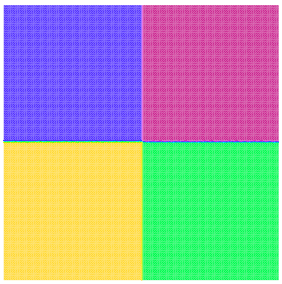
(a) Two sub-domains.



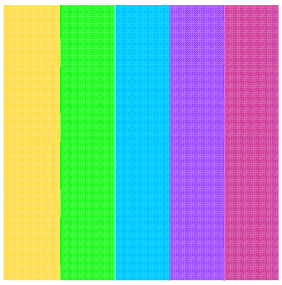
(b) Three sub-domains.



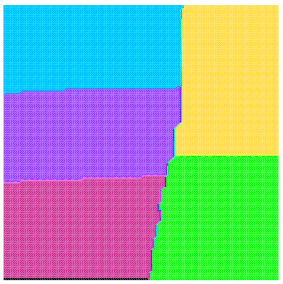
(c) Four sub-domains



(d) Four sub-domains.

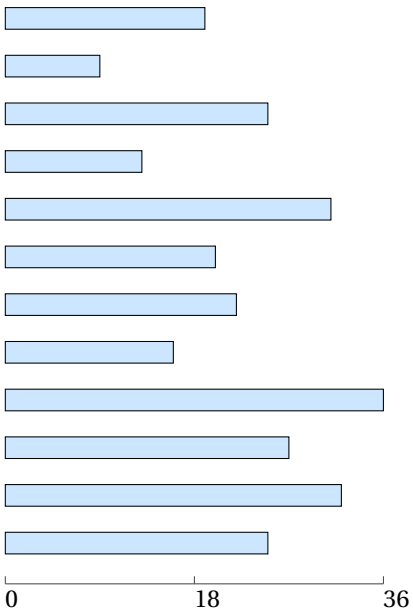


(e) Five sub-domains.

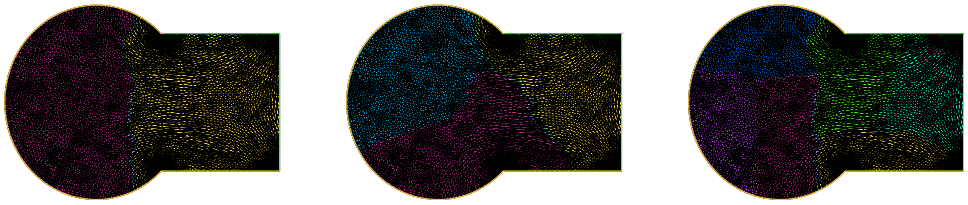


(f) Five sub-domains. (SCOTCH partitioner)

Method	$\kappa_{\approx}$	*n-iter*
(a) ASM	46.97	19
(a) MSM + EDOIC(5)	3.34	9
(b) ASM	74.19	25
(b) MSM + EDOIC(5)	4.47	13
(c) ASM	105.28	31
(c) MSM + EDOIC(5)	20.76	20
(d) ASM	77.76	22
(d) MSM + EDOIC(5)	31.98	16
(e) ASM	133.26	36
(e) MSM + EDOIC(5)	57.59	27
(f) ASM	102.06	32
(f) MSM + EDOIC(5)	50.55	25

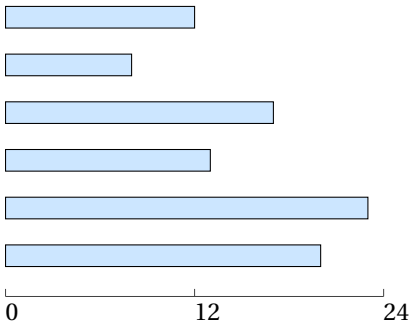


Experiment with complex domain



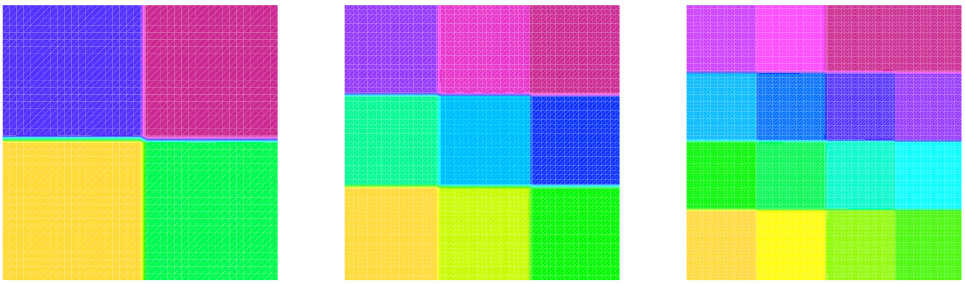
(a) Two sub-domains. (b) Three sub-domains. (c) Six sub-domains.

Method	$\kappa_{\approx}$	*n-iter*
(a) ASM	9.33	12
(a) MSM+EDOIC(3)	2.56	8
(b) ASM	13.79	17
(b) MSM+EDOIC(3)	7.67	13
(c) ASM	19.58	23
(c) MSM+EDOIC(3)	14.71	20



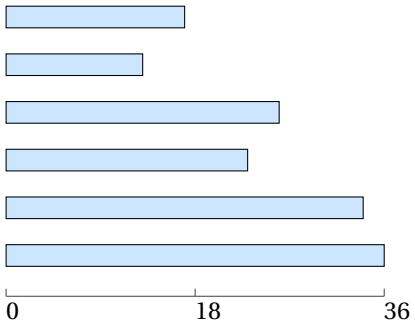
In this variant of experiment we keep the resolution of each subdomains fixed i.e, we use a global resolution $N_x \times N_y$, with decomposition into $M_x \times M_y$ subdomains, each of resolution $n_x \times n_y$. Therefore, $N_x = n_x M_x$ and $N_y = n_y M_y$.

Fixed size problem $n_x = n_y = 50$



(a) $M_x = M_y = 2$ (b) $M_x = M_y = 3$ (c) $M_x = M_y = 4$

Method	$\kappa_{\approx}$	*n-iter*
(a) ASM	32.97	17
(a) MSM+EDOIC(3)	18.21	13
(b) ASM	66.03	26
(b) MSM+EDOIC(3)	35.30	23
(c) ASM	112.76	34
(c) MSM+EDOIC(3)	96.97	36



Comments 3.4.2. *We see that for 4 or 9 sub-domains, the EDOIC Interface conditions automatically improves the convergence. But for more subdomains, the improvement is margined. This motivate the next chapter devoted to coarse grid corrections which are well suited to the many sub-domain case.*

Chapter 4

Two level method

In this chapter I would like to define a new way for algebraic construction of two-level preconditioner in which a coarse grid construction is based on approximated eigenvectors extracted from the Krylov space.

As we briefly mentioned in §1.4, in order to prevent stagnation in the convergence of the "one-level" domain decomposition methods for highly decomposed domain. One needs to define a two-level method to have a scalable algorithm i.e., an algorithm whose convergence rate is weakly dependent on the number of subdomains [58].

4.1 Abstract Preconditioner

Two-level domain decomposition methods are closely related to multigrid methods and deflation corrections. These methods are defined by two ingredients: a full rank matrix $Z \in \mathbb{R}^{n\times k}$ with $k << n$ and an algebraic formulation of the correction which implies solving a reduced size problem of order $k \times k$ called a coarse grid problem.

As we presented in §1.4, we can combine those elements with arbitrary preconditioner M^{-1} in order to construct abstract preconditioners like $\mathscr{P}_{as}$ (1.20), $\mathscr{P}_{bNtN}$ (1.21) and $\mathscr{P}_{a-def2}$ (1.22).

Notation 4.1 (Elements of abstract preconditioner)**.** *For the sake of simplicity we introduce here some notations:*

$\widetilde{A}$	*Coefficient matrix of the inflated linear system* $\widetilde{A}\widetilde{U} = \widetilde{F}$.
Z	*Full rank matrix which spans the coarse grid subspace.*
$E = Z^T\widetilde{A}Z$	*Coarse-grid matrix.*
$\Xi = ZE^{-1}Z^T$	*Coarse-grid correction matrix.*

$$Q_D = \mathbb{I} - \Xi\widetilde{A} = \mathbb{I} - Z(Z^T\widetilde{A}Z)^{-1}Z^T\widetilde{A}$$

Our choice of abstract preconditioner is the symmetric version of $\mathscr{P}_{a-def2}$ adapted to Modified Schwarz Method (see §2.5). Hence, using notation 4.1 we can define it as follows:

Two-level preconditioner $\mathscr{P}_{2lvl}$

$$\mathscr{P}_{2lvl} = (Q_D + \Xi)\,\widetilde{M}^{-1} \tag{4.1}$$

Operator $\widetilde{M}^{-1}$ refers to the *modified Schwarz method* (§2.5) preconditioner. Therefore, following example depicted on figure 2.9 (p. 58), we can extract from the modified operator (2.15) the following block preconditioner:

$$\widetilde{M}^{-1} = \begin{bmatrix} \widetilde{M}^{-1}_{\widetilde{D}_{\Omega_1}} & 0 & 0 \\ 0 & \widetilde{M}^{-1}_{\widetilde{D}_{\Omega_2}} & 0 \\ 0 & 0 & \widetilde{M}^{-1}_{\widetilde{D}_{\Omega_3}} \end{bmatrix}, \tag{4.2}$$

where

$$\widetilde{M}^{-1}_{\widetilde{D}_{\Omega_1}} = \begin{bmatrix} A_{1_i 1_i} & A_{1_i \Gamma_1^2} & 0 \\ A_{\Gamma_1^2 1_i} & A_{\Gamma_1^2 \Gamma_1^2} & A_{\Gamma_1^2 \Gamma_2^1} \\ 0 & A_{\Gamma_2^1 \Gamma_1^2} & A_{\Gamma_2^1 \Gamma_2^1} + S_2^1 \end{bmatrix}^{-1}$$

$$\widetilde{M}^{-1}_{\widetilde{D}_{\Omega_2}} = \begin{bmatrix} A_{2_i 2_i} & A_{2_i \Gamma_2^1} & A_{2_i \Gamma_2^3} & 0 & 0 \\ A_{\Gamma_2^1 2_i} & A_{\Gamma_2^1 \Gamma_2^1} & 0 & A_{\Gamma_2^1 \Gamma_1^2} & 0 \\ A_{\Gamma_2^3 2_i} & 0 & A_{\Gamma_2^3 \Gamma_2^3} & 0 & A_{\Gamma_2^3 \Gamma_3^2} \\ 0 & A_{\Gamma_1^2 \Gamma_2^1} & 0 & A_{\Gamma_1^2 \Gamma_1^2} + S_1^2 & 0 \\ 0 & 0 & A_{\Gamma_3^2 \Gamma_2^3} & 0 & A_{\Gamma_3^2 \Gamma_3^2} + S_3^2 \end{bmatrix}^{-1}$$

$$\widetilde{M}^{-1}_{\widetilde{D}_{\Omega_3}} = \begin{bmatrix} A_{3_i 3_i} & A_{3_i \Gamma_3^2} & 0 \\ A_{\Gamma_3^2 3_i} & A_{\Gamma_3^2 \Gamma_3^2} & A_{\Gamma_3^2 \Gamma_2^3} \\ 0 & A_{\Gamma_2^3 \Gamma_3^2} & A_{\Gamma_2^3 \Gamma_2^3} + S_2^3 \end{bmatrix}^{-1}.$$

Remark 4.1 (MSM *vs.* ASM)**.** *Using new interface conditions like* Sparse Patch *(see §2.6) or* EDOIC *(see chapter 3) is another strategy to accelerate the convergence. Therefore, we designed our method in such a way that it can benefit from modified Schwarz method i.e., from inflated operator extended by small interface sub-operators. This is a general solution because we can easily simplify MSM algorithm to additive Schwarz method (ASM) by discarding additional interface operators e.g., for* $S_2^1 = S_1^2 = S_3^2 = S_2^3 = 0$*, operator* (4.2) *is an ASM preconditioner.*

The second term ($\widetilde{M}^{-1}$) in two-level preconditioner (4.1) is a fine grid solver which can remove the very large eigenvalues of the coefficient matrix, which correspond to high frequency modes. But the small eigenvalues can still exist in the spectrum since they correspond to low frequency modes and represent certain global information. Therefore, we need a suitable the coarse solver ($Q_D + \Xi$) to efficiently deal with them.

The robustness of two-level preconditioners strongly depends on the choice of a coarse grid subspace. We focus now on the construction of the coarse space Z.

4.2 The Coarse Grid Space Construction

From deflation techniques [23, 47, 57] we know that it is preferable to choose the coarse grid subspace Z which consists of eigenvectors associated with the small eigenvalues. But the lower part of the spectrum of matrix $\widetilde{\mathrm{M}}^{-1}\widetilde{\mathrm{A}}$ can be very costly to obtain. However at the step m of the Krylov method we can use Krylov subspace
$\mathcal{K}_m(\widetilde{\mathrm{M}}^{-1}\widetilde{\mathrm{A}}, r_0) = \mathrm{SPAN}\{r_0, \widetilde{\mathrm{M}}^{-1}\widetilde{\mathrm{A}}r_0, \widetilde{\mathrm{M}}^{-1}\widetilde{\mathrm{A}}^2 r_0, \ldots, \mathrm{A}^{m-1}r_0\}$ in order to approximate the selected eigenvectors via a procedure described in the previous chapter (§3.2 p. 72). Thus, we can perform m iterations of Krylov-type solver for preconditioned system $\widetilde{\mathrm{M}}^{-1}\widetilde{\mathrm{A}}$ in order to approximate $n_{\mathcal{V}} \leq m$ eigenvectors $\mathcal{V}_i$ i.e.,

$$\begin{aligned} \left(\widetilde{\mathrm{M}}^{-1}\widetilde{\mathrm{A}}\right)\mathcal{V}_1 &\simeq \lambda_1 \mathcal{V}_1 \\ &\vdots \\ \left(\widetilde{\mathrm{M}}^{-1}\widetilde{\mathrm{A}}\right)\mathcal{V}_{n_{\mathcal{V}}} &\simeq \lambda_{n_{\mathcal{V}}} \mathcal{V}_{n_{\mathcal{V}}}. \end{aligned}$$

Where $(\lambda_i)_{1\leq i\leq n_{\mathcal{V}}}$ are the smallest (see remark 4.2) eigenvalues of the current square upper Hessenberg matrix H_m (3.19).

Remark 4.2 (Number of approximated eigenvectors.)**.** *The value $n_{\mathcal{V}}$ is an important parameter in our method. Along with the number of sub-domains it determines the size of the coarse space (see end of this section). The choice of $n_{\mathcal{V}}$ can be fixed (e.g., $n_{\mathcal{V}} = 2$ means: "compute two approximated eigenvectors associated with "two" smallest eigenvalues ($|\mathrm{Re}(\lambda_i)| \leq t_{\mathcal{V}}$)") or we can dynamically drive this value by some threshold $t_{\mathcal{V}}$ i.e., we compute all approximated eigenvectors $\mathcal{V}_i$ for which corresponding eigenvalue $|\mathrm{Re}(\lambda_i)| \leq t_{\mathcal{V}}$.*

Approximated eigenvectors $\mathcal{V}_i$ are of the same size as inflated domain $\widetilde{\mathrm{D}}_\Omega$ (see §2.3 p. 45). Moreover they obey partition of $\widetilde{\mathrm{D}}_\Omega$ in such a way that we can easily decompose them into

$$\mathcal{V}_i = \begin{bmatrix} [\mathcal{V}_i]_{\widetilde{\mathrm{D}}_{\Omega_1}} \\ [\mathcal{V}_i]_{\widetilde{\mathrm{D}}_{\Omega_2}} \\ \vdots \\ [\mathcal{V}_i]_{\widetilde{\mathrm{D}}_{\Omega_\mathrm{N}}} \end{bmatrix}.$$

where N is the number of subdomains and $[\mathcal{V}_i]_{\widetilde{\mathrm{D}}_{\Omega_j}}$ is the local contribution of $\mathcal{V}_i$ to subdomain $\widetilde{\mathrm{D}}_{\Omega_j}$ $(1 \leq j \leq \mathrm{N})$. Therefore, we can compose the following block operator from $n_{\mathcal{V}}$ approximated eigenvectors:

$$\mathrm{Z}^\star := \left[\mathcal{V}_1\, \mathcal{V}_2 \cdots \mathcal{V}_{n_{\mathcal{V}}}\right] = \begin{bmatrix} [\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_1}} & [\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_1}} & \cdots & [\mathcal{V}_{n_{\mathcal{V}}}]_{\widetilde{\mathrm{D}}_{\Omega_1}} \\ [\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_2}} & [\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_2}} & \cdots & [\mathcal{V}_{n_{\mathcal{V}}}]_{\widetilde{\mathrm{D}}_{\Omega_2}} \\ \vdots & \vdots & & \vdots \\ [\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_\mathrm{N}}} & [\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_\mathrm{N}}} & \cdots & [\mathcal{V}_{n_{\mathcal{V}}}]_{\widetilde{\mathrm{D}}_{\Omega_\mathrm{N}}} \end{bmatrix}$$

Unfortunately, vectors $(\mathcal{V}_i)_{1\leq i\leq n_{\mathcal{V}}}$ already belong to the Krylov space which we use in searching the solution of the system $\widetilde{\mathrm{M}}^{-1}\widetilde{\mathrm{A}}\widetilde{\mathrm{U}} = \widetilde{\mathrm{F}}$. For this reason using $\mathrm{Z}^\star$ in constructing a two-level preconditioner (4.1) will bring no benefits.

However we can apply a part wise splitting to $Z^\star$ in order to construct a coarse space which is similar in structure to (1.23) i.e., to the deflation subspace Z proposed by Nicolaides in [47]. Therefore, we can propose the following form of operator Z:

Coarse grid subspace Z

$$Z := \begin{bmatrix} [\mathcal{V}_1]_{\widetilde{D}_{\Omega_1}} & [\mathcal{V}_2]_{\widetilde{D}_{\Omega_1}} & \cdots & [\mathcal{V}_{n_\mathcal{V}}]_{\widetilde{D}_{\Omega_1}} & 0 & 0 & \cdots & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & [\mathcal{V}_1]_{\widetilde{D}_{\Omega_2}} & [\mathcal{V}_2]_{\widetilde{D}_{\Omega_2}} & \cdots & [\mathcal{V}_{n_\mathcal{V}}]_{\widetilde{D}_{\Omega_2}} & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots & & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & \cdots & [\mathcal{V}_1]_{\widetilde{D}_{\Omega_N}} & [\mathcal{V}_2]_{\widetilde{D}_{\Omega_N}} & \cdots & [\mathcal{V}_{n_\mathcal{V}}]_{\widetilde{D}_{\Omega_N}} \end{bmatrix}$$

In contrary to $Z^\star$, columns of Z are vectors which do not necessary belong to the Krylov space $\mathcal{K}_m(\widetilde{M}^{-1}\widetilde{A}, r_0) = \mathrm{SPAN}\{r_0, \widetilde{M}^{-1}\widetilde{A}r_0, (\widetilde{M}^{-1}\widetilde{A})^2 r_0, \ldots, (\widetilde{M}^{-1}\widetilde{A})^{m-1} r_0\}$, thus there is some benefit we can incorporate to the robustness of two-level preconditioner (4.1) with Z defined in this way. Moreover, part wise, block structure of Z is very suitable for parallel implementation.

We can also easily determine the size of the coarse space. Since $Z \in \mathbb{R}^{n \times (n_\mathcal{V} N)}$, the coarse problem to solve will be of size $n_\mathcal{V} N \times n_\mathcal{V} N$.

4.3 Parallel implementation

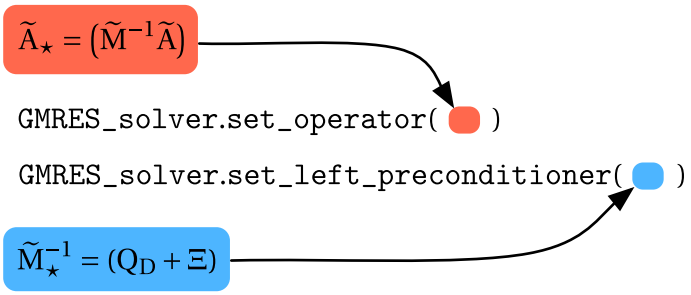
The main challenge during implementation of two-level preconditioner in form (4.1) is to encapsulate its complex structure in one object of the same type as `Preconditioner` class in ADDMlib (see §2.2.3). In other words, from the user point of view, a two-level preconditioner must be a matrix operator which can act on `DDMVectors` via `axpy` routine, where `axpy` is a method of any matrix type object in ADDMlib to perform matrix-vector product. This is, for matrix A and two vectors V_1 and V_2, we can define:

$$\left[\mathrm{A.axpy}(V_1, V_2)\right] \quad := \quad [V_2 = V_2 + AV_1]\,. \tag{4.3}$$

Matrix-vector products are essential for building Krylov spaces. Therefore, if we want to build Krylov space for inflated system $\widetilde{A}\widetilde{U} = \widetilde{F}$ preconditioned by (4.1) we need to be able to perform a following computation:

$$V_1 = \left[(Q_D + \Xi)\left(\widetilde{M}^{-1}\widetilde{A}\right)\right] V_2. \tag{4.4}$$

In order to fit this computation to the simple interface of linear algebra kernel in ADDMlib, we apply the following decomposition:



Therefore, if we define matrix-vector product routines for $\widetilde{M}_\star^{-1}$ and $\widetilde{A}_\star$, iterative solvers implemented in ADDMlib will be able to create a Krylov space of the form

$$\mathcal{K}_m(\widetilde{M}_\star^{-1}\widetilde{A}_\star, r_0) = \text{SPAN}\left\{r_0, \widetilde{M}_\star^{-1}\widetilde{A}_\star r_0, (\widetilde{M}_\star^{-1}\widetilde{A}_\star)^2 r_0, \ldots, (\widetilde{M}_\star^{-1}\widetilde{A}_\star)^{m-1} r_0\right\}, \tag{4.5}$$

in order to find a solution $\widetilde{U}$ of the system $\widetilde{M}_\star^{-1}\widetilde{A}_\star\widetilde{U} = \widetilde{F}$.

4.3.1 Matrix-vector product for compose operator $\widetilde{A}_\star$

Matrix-vector product between the operator $\widetilde{A}_\star$ and `DDMVector` V_i is easy to perform since `axpy` methods of its components are well defined and belong to linear algebra kernel of ADDMlib (see §2.2). For that reason, we can propose the following simple algorithm:

Algorithm 6 $\widetilde{A}_\star$.axpy(V_1, V_2)

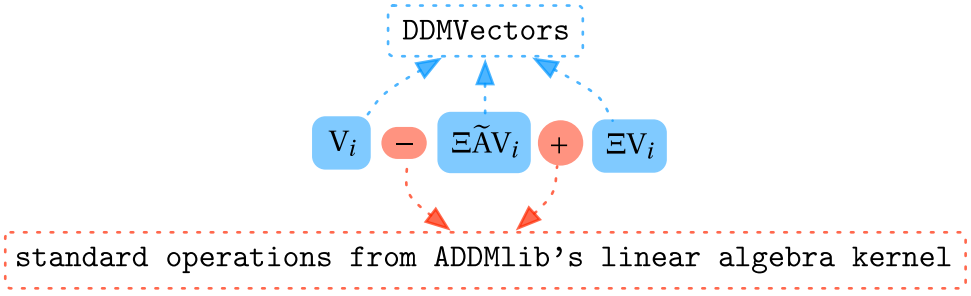
1: `temporary_vector` $:= \widetilde{A}V_1$
2: $V_2 = V_2 + \widetilde{M}^{-1}$`temporary_vector`

4.3.2 Matrix-vector product for preconditioner $\widetilde{M}_\star^{-1}$

In case of the matrix-vector product for the operator $\widetilde{M}_\star^{-1}$ the "axpy" routine is more difficult to implement, since $\widetilde{M}_\star^{-1}$ is an abstract container for "interaction" between three different operators (Z, Z^T and $\widetilde{A}$). However if we write $\widetilde{M}_\star^{-1}V_i$ explicitly we can apply similar decomposition as for (4.4):

$$(Q_D + \Xi)V_i \quad = \quad \left[\left(\mathbb{I} - \Xi\widetilde{A}\right) + \Xi\right]V_i \quad = \quad V_i - \Xi\widetilde{A}V_i + \Xi V_i \tag{4.6}$$

where



Therefore, only ΞV_i is an operation which must be implemented additionally.

4.3.3 Coarse grid correction - Ξ

The essence of evaluation ΞV_i is in solving coarse problem $E^{-1}\,(\texttt{times})\,(Z^T V_1)$. In order to create coarse operator $E = Z^T \widetilde{A} Z$ we need to perform two matrix-matrix multiplications ($\widetilde{B} = \widetilde{A}Z$ and $Z^T\widetilde{B}$), where Z is a part-wise operator with the block structure defined in previous section. Operator Z originates from $Z^\star$ which is a collection of `DDMVectors`, thus one way to evaluate $\widetilde{A}Z$ is to perform matrix-vector products between operator $\widetilde{A}$ and columns of $Z^\star$ with some mask.

In chapter §2 we defined an `axpy` method (realisation of matrix-vector product) for global `DDMOperator` and `DDMVector` which we expressed in the form of linear combination of matrix-vector products between `PartialOperators` and `PartialVectors`. We can exploit this decomposition and create a new specialised object: `SparseVectorCollection` (`SVC`) which is a collection of `PartialVectors` uniquely defined by two numbers: `PartID` and index $1 \leq j^{\mathrm{SP}} \leq n_{\mathcal{V}}\mathrm{N}$, witch denotes the virtual column. `Partial Vectors` are distributed along process in the same manner as `Partial Vectors` in `DDMVector` i.e., all `Partial Vectors` with the same `PartID` are stored on the same process.

Using `SVC` we can easily construct operator Z from $Z^\star$ by associating each `Partial Vector` $\in Z^\star$ with proper j^{SP}. For example in three sub-domain case and two `DDMVectors` in $Z^\star$ we have:

$$Z^\star := \begin{bmatrix} [\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_1}} & [\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_1}} \\ [\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_2}} & [\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_2}} \\ [\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_3}} & [\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_3}} \end{bmatrix} \tag{4.7}$$

$$Z := \begin{bmatrix} [\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_1}} & [\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_1}} & & & & \\ & & [\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_2}} & [\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_2}} & & \\ & & & & [\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_3}} & [\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_3}} \end{bmatrix} \tag{4.8}$$

$$[\mathtt{SVC}]\,Z := \begin{bmatrix} (1)\left[[\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_1}}\right]_1 & (1)\left[[\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_1}}\right]_2 \\ (2)\left[[\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_2}}\right]_3 & (2)\left[[\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_2}}\right]_4 \\ (3)\left[[\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_3}}\right]_5 & (3)\left[[\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_3}}\right]_6 \end{bmatrix}. \tag{4.9}$$

Where [SVC]Z is a `SparseVectorCollection` for Z in which each `PrtialVecotr` is denoted by (`PartID`)$[\mathcal{V}]_{j^{\mathrm{SP}}}$.

In order to successfully use this new data structure in (4.6) we need to define couple of algebraic operations in form **a** (`times`) **b** = **c** (see table 4.1).

Operation [`DDMOperator`][`SVC`] = [`SVC`]

Operation [`DDMOperator`][`SVC`] is a generalisation of `axpy` method for `DDMOperator` and `DDMVector` described in details in §2.2.2 (p. 43). The main difference is an additional index j^{SP}, which ensures proper structure of final operator (of type [`SVC`]) which is a collection of partial results i.e., results originate from evaluation `axpy` method between `Partial`

	data type	**var a**	*data type*	**var b**		*data type*	**var c**
1)	[DDMOperator]	$\widetilde{\mathrm{A}}$	[SVC]	Z_1	=	[SVC]	Z_2
2)	[SVC]	Z^{T}	[DDMVector]	$\mathcal{W}$	=	[small vector $\in \mathbb{R}^{n_\mathcal{V}\mathrm{N}}$]	v
3)	[SVC]	Z	[small vector $\in \mathbb{R}^{n_\mathcal{V}\mathrm{N}}$]	v	=	[DDMVector]	$\mathcal{V}$
4)	[SVC]	$\mathrm{Z}_1^{\mathrm{T}}$	[SVC]	Z_2	=	[small sparse matrix $\in \mathbb{R}^{n_\mathcal{V}\mathrm{N}\times n_\mathcal{V}\mathrm{N}}$]	E

Table 4.1: List of algebraic operation for SVC object defined in ADDMlib. All operations are in form **a** (times) **b** = **c**.

Operators in DDMOperator and Partial Vectors in Sparse Vector Collection container. Following our example with three sub-domain case we have:

$$
\begin{aligned}
\widetilde{\mathrm{A}}\mathrm{Z} &= \left[\begin{array}{ccc|cc|c} \mathrm{A}_{11} & \mathrm{A}_{12} & \mathrm{A}_{13} & & & \\ \hline \mathrm{A}_{21} & \mathrm{A}_{22} & & & & \\ \hline \mathrm{A}_{31} & & \mathrm{A}_{33} & & & \end{array}\right] \left[\begin{array}{cc|cc|cc} [\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_1}} & [\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_1}} & & & & \\ \hline & & [\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_2}} & [\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_2}} & & \\ \hline & & & & [\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_3}} & [\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_3}} \end{array}\right] \\
&= \left[\begin{array}{cccccc} \mathrm{A}_{11}[\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_1}} & \mathrm{A}_{11}[\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_1}} & \mathrm{A}_{12}[\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_2}} & \mathrm{A}_{12}[\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_2}} & \mathrm{A}_{13}[\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_3}} & \mathrm{A}_{13}[\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_3}} \\ \hline \mathrm{A}_{21}[\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_1}} & \mathrm{A}_{21}[\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_1}} & \mathrm{A}_{22}[\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_2}} & \mathrm{A}_{22}[\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_2}} & & \\ \hline \mathrm{A}_{31}[\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_1}} & \mathrm{A}_{31}[\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_1}} & & & \mathrm{A}_{33}[\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_3}} & \mathrm{A}_{33}[\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_3}} \end{array}\right] \\
&= \left[\begin{array}{cccccc} (1)\mathcal{V}_1 & (1)\mathcal{V}_2 & (1)\mathcal{V}_3 & (1)\mathcal{V}_4 & (1)\mathcal{V}_5 & (1)\mathcal{V}_6 \\ \hline (2)\mathcal{V}_1 & (2)\mathcal{V}_2 & (2)\mathcal{V}_3 & (2)\mathcal{V}_4 & & \\ \hline (3)\mathcal{V}_1 & (3)\mathcal{V}_2 & & & (3)\mathcal{V}_5 & (3)\mathcal{V}_6 \end{array}\right] \begin{array}{l} \leftarrow \texttt{Proc 0} \\ \leftarrow \texttt{Proc 1} \\ \leftarrow \texttt{Proc 2} \end{array}
\end{aligned}
$$

Remark 4.3. *In example in this section we assume that each part is associated with different process. We display this by horizontal (or vertical in case of transposed operators) lines in operators representation. We put also some restrictions on distribution of* Partial Operators *in* DDMOperator $\widetilde{\mathrm{A}}$ *i.e., each* Parital Operator *with the same* PartOutID *is stored on the same process. This restriction indicates that operation "*[DDMOperator][SVC]*" needs some data transfer between processes. Therefore in order to perform locally all "*axpyies*" between* Partial Operators *and* Partial Vector *in* SVC *we need to transfer (via point-to-point communication) sparse data from* SVC *to* DDMOperator *i.e., sparse parts of* Partial Vectors *in* SVC *accordingly to non-zero columns in endomorphic* Partial Operators *in* DDMOperator.

Operation [SVC]$^{\mathrm{T}}$[DDMVector] = [$v \in \mathbb{R}^{n_\mathcal{V}\mathrm{N}}$]

Operation [SVC]$^{\mathrm{T}}$[DDMVector] creates local copy of coarse vector $v \in \mathbb{R}^{n_\mathcal{V}\mathrm{N}}$ on each process. We exploit in this operation assumption that all Partial Vectors with the same PartID are of the same size, thus in order to fill vector v, we need perform series of dot products i.e.:

$$
v_{j^{\mathrm{SP}}} = \sum_{\mathtt{PartID}} [(\mathtt{PartID})\mathcal{V}_{j^{\mathrm{SP}}}] \cdot [\mathcal{W}]_{\widetilde{\mathrm{D}}_{\Omega_{\mathtt{PartID}}}} \tag{4.10}
$$

where [(PartID)$\mathcal{V}_{j^{\mathrm{SP}}}$] are elements of SVC and $[\mathcal{W}]_{\widetilde{\mathrm{D}}_{\Omega_{\mathtt{PartID}}}}$ are Partial Vectors in global DDMVector $\mathcal{W}$.

In practice on each process we create temporary $v_{\mathrm{TMP}} \in \mathbb{R}^{n_{\mathcal{V}}\mathrm{N}}$ which we partially fill by performing all local dot products (all Partial Vectors with the same PartID in SVC and DDMVectors are stored on the same process). Then we sum those partial results by calling MPI_ALLREDUCE in order to create final result on each process.

Following our three sub-domains example from previous subsections we have:

$$
\begin{aligned}
Z^{\mathrm{T}}\mathcal{W} &= \left[\begin{array}{c|c|c} [\mathcal{V}_1]^{\mathrm{T}}_{\widetilde{\mathrm{D}}_{\Omega_1}} & & \\ [\mathcal{V}_2]^{\mathrm{T}}_{\widetilde{\mathrm{D}}_{\Omega_1}} & & \\ & [\mathcal{V}_1]^{\mathrm{T}}_{\widetilde{\mathrm{D}}_{\Omega_2}} & \\ & [\mathcal{V}_2]^{\mathrm{T}}_{\widetilde{\mathrm{D}}_{\Omega_2}} & \\ & & [\mathcal{V}_1]^{\mathrm{T}}_{\widetilde{\mathrm{D}}_{\Omega_3}} \\ & & [\mathcal{V}_2]^{\mathrm{T}}_{\widetilde{\mathrm{D}}_{\Omega_3}} \end{array}\right] \left[\begin{array}{c} [\mathcal{W}]_{\widetilde{\mathrm{D}}_{\Omega_1}} \\ \hline [\mathcal{W}]_{\widetilde{\mathrm{D}}_{\Omega_2}} \\ \hline [\mathcal{W}]_{\widetilde{\mathrm{D}}_{\Omega_3}} \end{array}\right] = \\
&= \underbrace{\left[\begin{array}{c} [\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_1}} \cdot [\mathcal{W}]_{\widetilde{\mathrm{D}}_{\Omega_1}} \\ [\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_1}} \cdot [\mathcal{W}]_{\widetilde{\mathrm{D}}_{\Omega_1}} \\ \hline 0 \\ 0 \\ \hline 0 \\ 0 \end{array}\right] + \left[\begin{array}{c} 0 \\ 0 \\ \hline [\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_2}} \cdot [\mathcal{W}]_{\widetilde{\mathrm{D}}_{\Omega_2}} \\ [\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_2}} \cdot [\mathcal{W}]_{\widetilde{\mathrm{D}}_{\Omega_2}} \\ \hline 0 \\ 0 \end{array}\right] + \left[\begin{array}{c} 0 \\ 0 \\ \hline 0 \\ 0 \\ \hline [\mathcal{V}_1]_{\widetilde{\mathrm{D}}_{\Omega_3}} \cdot [\mathcal{W}]_{\widetilde{\mathrm{D}}_{\Omega_3}} \\ [\mathcal{V}_2]_{\widetilde{\mathrm{D}}_{\Omega_3}} \cdot [\mathcal{W}]_{\widetilde{\mathrm{D}}_{\Omega_3}} \end{array}\right]}_{\texttt{MPI\_ALLREDUCE(..., ..., ..., ..., MPI\_SUM, ...)}} = \left[\begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{array}\right].
\end{aligned}
$$

Operation [SVC][$v \in \mathbb{R}^{n_{\mathcal{V}}\mathrm{N}}$] = [DDMVector]

Since each process has is own copy of vector v (see previous subsection), operation [SVC] [$v \in \mathbb{R}^{n_{\mathcal{V}}\mathrm{N}}$] do not involve any communication. In order to calculate components of DDMVector we need to sum Partial Vectors with the same PartID in SVC scaled by proper value from v i.e.:

$$
[\mathcal{W}]_{\widetilde{\mathrm{D}}_{\Omega_{\mathtt{PartID}}}} = \sum_{j^{\mathrm{SP}}} v_{j^{\mathrm{SP}}} [(\mathtt{PartID})\mathcal{V}_{j^{\mathrm{SP}}}] \tag{4.11}
$$

where [(PartID)$\mathcal{V}_{j^{\mathrm{SP}}}$] are elements of SVC, $[\mathcal{W}]_{\widetilde{\mathrm{D}}_{\Omega_{\mathtt{PartID}}}}$ is a component of DDMvector and $v_{j^{\mathrm{SP}}}$ is an value from vector v.

Operation $[\mathrm{SVC}]^{\mathrm{T}}[\mathrm{SVC}] = [\mathrm{E} \in \mathbb{R}^{n_{\mathcal{V}}\mathrm{N} \times n_{\mathcal{V}}\mathrm{N}}]$

The role of operation $[\mathrm{SVC}]^{\mathrm{T}}[\mathrm{SVC}]$ is a creation of coarse operator $\mathrm{E} \in \mathbb{R}^{n_{\mathcal{V}}\mathrm{N} \times n_{\mathcal{V}}\mathrm{N}}$ which must be duplicated on each process. This operation is a generalisation of $[\mathrm{SVC}]^{\mathrm{T}}$[DDMVector]. Therefore to compute values of E we also need to perform a series of dot products:

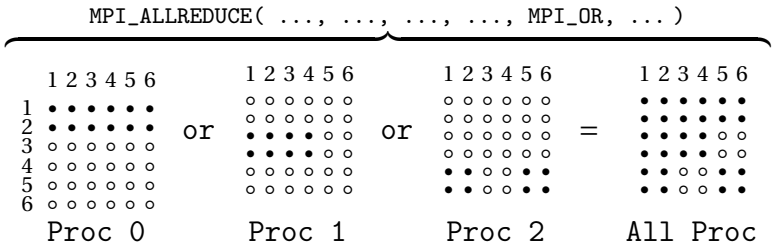
$$
e_{ij} = \sum_{\mathtt{PartID}} (\mathtt{PartID})[\mathcal{V}_1]_i \cdot (\mathtt{PartID})[\mathcal{V}_2]_j \tag{4.12}
$$

where (PartID)$[\mathcal{V}_1]_i$ are elements of [SVC] Z_1^T and (PartID)$[\mathcal{V}_1]_i$ of [SVC] Z_2 (see table 4.1).

In case of operation [SVC]^T[DDMVector] we first create temporary vectors $v_{\text{TMP}} \in \mathbb{R}^{n_\gamma \text{N}}$ for partial and local results. Next, we sum them together and final product is automatically created on each process.

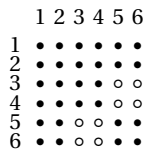
In order to adapt this procedure to more general operation described in this subsection, we need change temporary vectors to temporary operators. However we can not allocate them as a full matrixes due to memory limitation (see remark 4.4). Moreover we expect coarse operator to be sparse thus we prefer to store it in CSR format. Unfortunately we can not easily deduce CSR structure of E while components of Z_1^T and Z_2 are distributed. Thus first we create on each process temporary matrix $B_{tmp} \in \{\texttt{true}, \texttt{false}\}^{n_\gamma \text{N} \times n_\gamma \text{N}}$ which all values are set to `false`. Next we evaluate all local dot products between `Partial Vectors` with the same `PartID` from [SVC] Z_1 and [SVC] Z_2. Values obtained in this way are store in a map i.e., in a standard C++ container which is a sorted associative array of unique keys and associated data [1]. The unique key in this case is a pair of indices <i , j> i.e., a column indexes of the `Partial Vectors` from Z_1 and Z_2. For each non-zero value computed in this way, a value b_{ij} in temporary matrix B_{tmp} is changed to `true`.

When we collect locally all partial results we perform `MPI_ALLREDUCE( ..., MPI_OR)` for local matrixes B_{tmp}. In a result each process will have a copy of matrix B_{tmp} in which values `true` denote non-zeros in final matrix E of the same size. For our three sub-domain example we have:



Where • = `true`, ∘ = `false` and **or** [2] states for logical disjunction.

From final version of B_{tmp} we can easily construct CSR structure for operator E:



$$\rightarrow \text{(CSR)} \quad \begin{aligned} \text{rows} &= [1\ 7\ 13\ 17\ 21\ 25\ \mathbf{29}] \\ \text{cols} &= [1\,2\,3\,4\,5\,6\,1\,2\,3\,4\,5\,6\,1\,2\,3\,4\,1\,2\,3\,4\,1\,2\,5\,6\,1\,2\,5\,6] \\ \text{data} &= [e_{11}\ e_{12}\ e_{13}\ e_{14}\ e_{15}\ e_{16}\ e_{21}\ e_{22}\ e_{23}\ e_{24}\ e_{25}\ e_{26}\ \ldots\ e_{65}\ e_{66}] \end{aligned}$$

While converting B_{tmp} to CSR structure we fill data vector by our local results and in place of missing values we put zeros. In this way the data vector has the same length on each processor and we can perform another `MPI_ALLREDUCE( ..., MPI_SUM)` in order to join all distributed products and create final operator on each process.

1. For more details see [55].
2. In logic and mathematic, **or**, also know as logical Inclusive disjunction, is a logical operator that results in true whenever one or more of its operands are true.

$$\texttt{MPI\_ALLREDUCE}\begin{cases}[e_{11}\ e_{12}\ e_{13}\ e_{14}\ e_{15}\ e_{16}\ e_{21}\ e_{22}\ e_{23}\ e_{24}\ e_{25}\ e_{26}\ 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0] & \texttt{Proc 0}\\ [0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\ e_{31}\ e_{32}\ e_{33}\ e_{34}\ e_{41}\ e_{42}\ e_{43}\ e_{44}\ 0\,0\,0\,0\,0\,0\,0\,0] & \texttt{Proc 1}\\ [0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\ e_{51}\ e_{52}\ e_{55}\ e_{56}\ e_{61}\ e_{62}\ e_{65}\ e_{66}] & \texttt{Proc 2}\end{cases}$$

$$\text{data} = \underbrace{[e_{11}\ e_{12}\ e_{13}\ e_{14}\ e_{15}\ e_{16}\ e_{21}\ e_{22}\ e_{23}\ e_{24}\ e_{25}\ e_{26}\ e_{31}\ e_{32}\ e_{33}\ e_{34}\ e_{41}\ e_{42}\ e_{43}\ e_{44}\ e_{51}\ e_{52}\ e_{55}\ e_{56}\ e_{61}\ e_{62}\ e_{65}\ e_{66}]}_{\texttt{All Proc}}$$

Remark 4.4. *If we assume that matrix* E *is full and we skip procedure of prediction a sparsity we can easily ran out of memory. We can estimate size of matrix full of* `double` *(64* `bits`*) by following formula:*

$$\frac{\left[(n \text{ vectors in } Z^{\star})\,(\text{number of parts})\right]^2 64\,\texttt{bits}}{8\,388\,608\,\texttt{bits}} = [\mathit{matrix\ size}]\,\texttt{MB}. \tag{4.13}$$

Therefore in case of 1024 *subdomains and coarse operator builded from* 5 `DDMVectors` *we would need* 200 `MB`*. With* 3 *additional vectors we would reach* 512 `MB`*. For comparison in experiment with double inflated 3D Laplace case (see §5 p.116), coarse operator* E *builded from* 5 *eigenvectors for* 1024 *subdomain has only* 556600 *non-zero elements. Hence only* 4.25 `MB` *were needed to store its data in CSR format.*

Finally to find result of $E^{-1}v$, we solve small linear system using external direct solver like we perform in case of DDM preconditioners. Therefore, we refer to §2.2.3 for more details.

4.4 Numerical results

As we showed in previous sections, in order to define the two-level preconditioner (4.1) we need to set up the coarse space Z. More precisely, for each computation we need to fix up two parameters. The number of iteration of Krylov solver - m, after which we will compute $n_{\mathcal{V}} \leq m$ approximated eigenvectors. Therefore, in order to identify difference between numerical experiments with different values of m and $n_{\mathcal{V}}$ we propose a following notation

$$\texttt{P2Level}(m, n_{\mathcal{V}}).$$

In order to present influence of the new two-level preconditioner on condition number κ and number of iterations, we following numerical experiment which different configuration define separate subsections.

Numerical Experiment 4.1. *Let us consider Laplace equation* $-\eta\Delta(\mathbf{u}) = 0$*, discretized using P1-type finite elements on 2D or 3D domain* Ω *triangulated by the* Delaunay-Voronoi*-type algorithm with uniform Dirichlet condition on boundary (*$\mathbf{u} = 0$ *on* $\partial\Omega$*). The right hand side of resulting linear system is a function* f *which gives random values from set* $\langle 1,2)$ *(fixed for all variants of test). For domain decomposition into* N *sub-domain we use SCOTCH graph*

partitioner or we perform manual partition. We solve the resulting discrete system using GMRES preconditioned by a preconditioner specified by variant of experiment. The initial guess is chosen to be $\mathbf{u}^{(0)} = 0$ *and the stopping criterion* $\|r_i\| \leq tol \cdot \|r_0\|$ *for default* $tol = 1 \times 10^{-6}$ *(in 3D case we use also lower tolerance). The roughly estimated condition number is given as* $\kappa_{\approx} = \lambda_{max}/\lambda_{min}$ *where* $\lambda_{\{min,max\}}$ *are the approximated, extreme eigenvalues of* $(\widetilde{M}^{-1}\widetilde{A})$ *or* $(\widetilde{M}_{\star}^{-1}\widetilde{A}_{\star})$.

4.4.1 Successive and Adaptive two-level preconditioner

Since it is necessary to perform at least m iterations of the Krylov solver to approximate given number of eigenvectors, we can distinguish two tactics:

I **successive** - in which the first solver after m iteration computes approximated eigenvectors and then continues its iterations in order to obtain solution. After that we build two-level preconditioner using pre-calculated eigenvectors and we can use new type of preconditioning in second solve starting with default initial guess.

II **adaptive** - in which we **stop** first solver after m iterations and we compute approximated eigenvectors. Then we construct new preconditioner in order to use it in second solve which as a initial guess uses "imprecise" solution from the stopped, first solver.

Successive tactic (default for experiments in this chapter) refers to situations in which we need to solve couple of linear systems with the same operator or not too different. While the adaptive technique is more adequate for solving difficult linear systems with long stagnation in convergence.

4.4.2 How to read plots

We present all the results of our experiments in form of plots. Therefore, for each test we depicted number of iterations and estimated condition number of preconditioned system, in such a way that they can be easily compared with a reference solution (usually once inflated system preconditioned by additive Schwarz method) or with different variants of the same experiment. In case of experiment with Black-Oil for all tests we also depicted *convergence curves*[3] for both iterative solvers used in given variant. In case of adaptive tactic, the convergence curve of first solver (one with one-level preconditioner) is limited to the first m iterations.

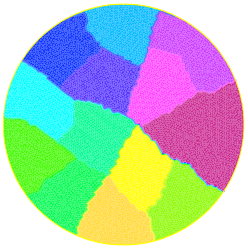


Figure 4.1: Unite disc

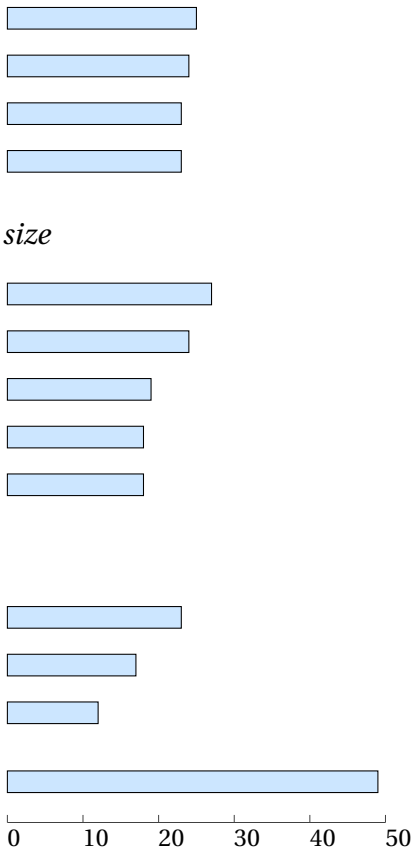
4.4.3 Two-level preconditioner *versus* quality of eigenvectors approximation and size of coarse space

In the following variant of experiments 4.1 we test how the quality of eigenvectors approximations and size of coarse space influences the robustness of the two-level precondi-

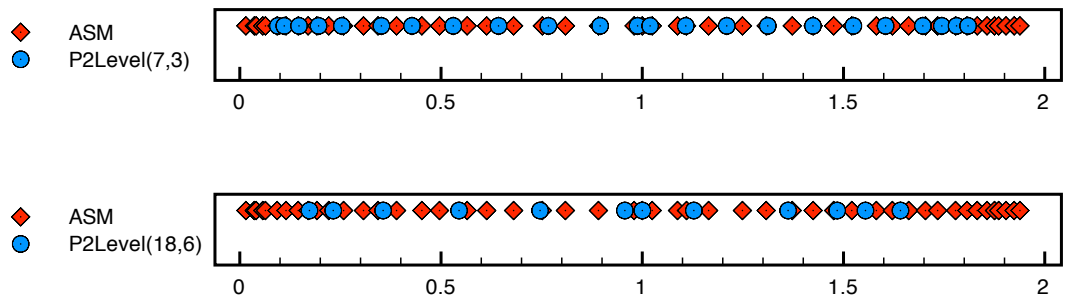
3. In our case it is a logarithm of residual norm ($p = 2$) for each iteration of Krylov solver.

tioner. The computational domain Ω is chosen to be triangulated unite disk (85306 triangles). For decomposition we use SCOTCH partitioner over adjacency graph originate from underlying matrix. The number of sub-domain is fixed to N = 14. Different variants of `P2Level`$(m, n_{\mathcal{V}})$ are compared with one-level (additive Schwarz method) reference solution.

Method	$\kappa_{\approx}$	*n-iter*
Different coarse space size		
P2Level(7,1)	26.95	25
P2Level(7,2)	19.69	24
P2Level(7,3)	19.13	23
P2Level(7,4)	19.51	23
Different upper Hessenberg matrix size		
P2Level(3,3)	26.50	27
P2Level(6,3)	21.39	24
P2Level(9,3)	15.55	19
P2Level(12,3)	15.83	18
P2Level(15,3)	14.96	18
Results for bigger coarse space		
P2Level(6,6)	17.29	23
P2Level(12,6)	14.83	17
P2Level(18,6)	9.48	12
(referece) ASM	128.00	49



Example spectra of $\widetilde{\mathrm{M}}_{\star}^{-1}\widetilde{\mathrm{A}}_{\star}$ *and* $\widetilde{\mathrm{M}}^{-1}\widetilde{\mathrm{A}}$ *(ASM)*



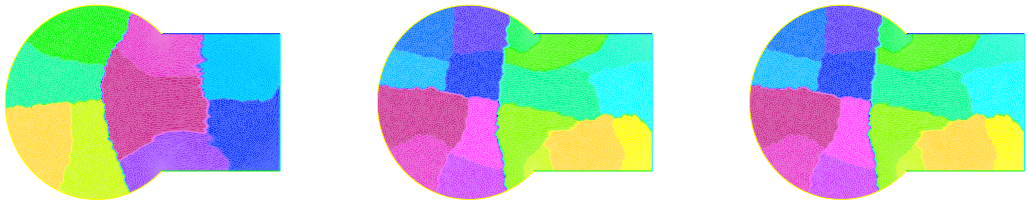
Comments 4.4.1. *Using the one-level method, we have 49 iterations. Using the two-level method brings benefits. As expected for large values of m and* $n_{\mathcal{V}}$ *the iteration counts are better and better. For example for* $m = 18$ *and* $n_{\mathcal{V}} = 6$ *we need only 12 iterations. Also if we consider*

upper Hessenberg matrix sizes with a coarse space of fixed size (3 × number of subdomains), we see that using large m brings benefit up to m = 9 but for large m there is no improvements. That is, for m = 9, the quality of the first 3 Ritz eigenvectors was sufficient. Now if we add to the iteration counts the value of m (the number of iterations of the first solve used to compute the Ritz eigenvectors) we see that the total iteration counts is nearly flat since it takes values between 29 and 33. In this case, the method is not too sensible to the choice of the parameters, if the coarse space has to be used for subsequent solves, it is better to use "large" values of m and n_γ since the first solve will not be penalised while it will more efficiently accelerate the subsequent solves.

4.4.4 Two-level preconditioner *versus* number of subdomains

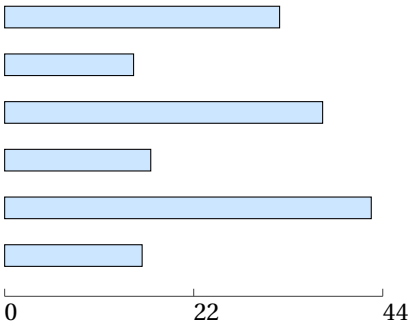
In the following variant of our experiments we present how the number and a shape of sub-domains influence the robustness of the two-level preconditioner. The computational domains Ω is chosen to be a complex domain which is an union of disc and rectangle. The mesh is of size 9000 triangles. We divided Ω into 9 and more sub-domains. Different partitions via graph partitioner are depicted above results.

Experiment with complex domain and fixed global mesh



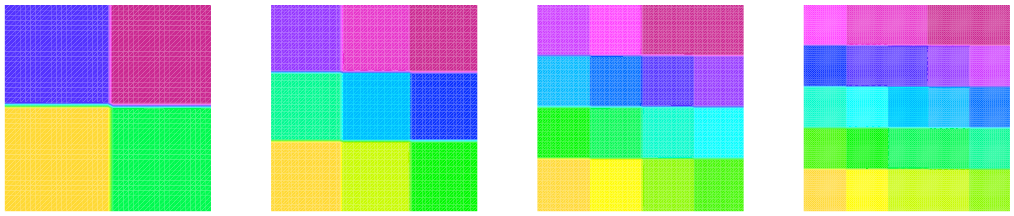
(a) 9 sub-domains. (b) 18 sub-domains. (c) 27 sub-domains.

Method	$\kappa_\approx$	*n-iter*
(a) ASM	52.05	32
(a) 2Level(6,3)	8.64	15
(b) ASM	80.42	37
(b) 2Level(6,3)	12.23	17
(c) ASM	98.66	43
(c) 2Level(6,3)	9.38	16



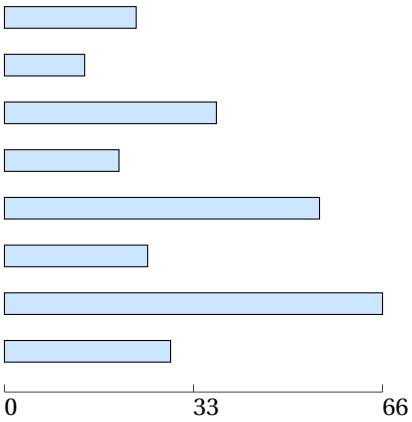
Comments 4.4.2. *Notice that in this case, the overlap has fixed physical size so that the increase in the iteration counts of the one level method are due only to the increase in the number of subdomains.*

In following experiments we keep the resolution of each subdomains fixed i.e, we use a global resolution $N_x \times N_y$, with decomposition into $M \times M$ subdomains, each of resolution $n \times n$, where $n = 70$. Therefore, $N_x = N_y = nM$. Therefore, the physical size of the overlap decreases while the number of subdomains increases.



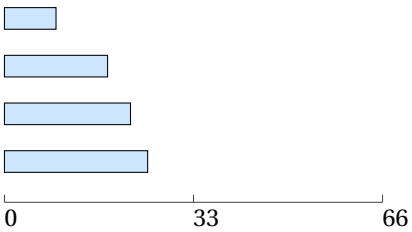
(a) M = 2 (b) M = 3 (c) M = 4 (d) M = 5

Method	$\kappa_{\approx}$	*n-iter*
(a) ASM	93.69	23
(a) 2Level(6,3)	10.05	14
(b) ASM	244.71	37
(b) 2Level(6,3)	17.35	20
(c) ASM	471.21	55
(c) 2Level(6,3)	29.10	25
(d) ASM	773.05	66
(d) 2Level(6,3)	32.56	29



Results for bigger coarse space

Method	$\kappa_{\approx}$	*n-iter*
(a) 2Level(10,4)	5.59	9
(b) 2Level(10,4)	18.61	18
(c) 2Level(10,4)	24.88	22
(d) 2Level(10,4)	24.70	25



Comments 4.4.3. *In these experiments, the increase of iteration counts for the two-level method comes from the fact that as M (number of sub-domains) increases, the physical size of the overlap decreases. In the next paragraph, Sparse Patch interface condition gives much better robustness to the two-level method in this case.*

4.4.5 Two-level preconditioner with Sparse Patch

Let us introduce some additional notation describing different type of algebraic techniques used in this experiments:

method name	***description***
ASM no I	*Additive Schwarz Method (see algorithm 3 p. 44) with minimal overlap (no inflation §2.3).*
ASM no I ($low\ tol$)	*Additive Schwarz Method with minimal overlap (no inflation). ($low\ tol$) stays for lower tolerance for GMRES, by default* $tol = 1e-6$*, but in this experiment* $tol = 1e-8$.
ASM + I(1)	*Additive Schwarz method with one level inflation:* $n_i ter(\kappa\widetilde{\mathrm{M}}^{-1}\widetilde{\mathrm{A}})$
ASM + I(2)	*Additive Schwarz Method with two level inflation.*
P2Level(m, n_V)	*Two level preconditioner where* $\widetilde{\mathrm{M}}^{-1}$ *=ASM.*
Patch(g, p, l_{max}, α)	*Modified Schwarz Method (MSM) (see §2.5) with Sparse Patch. (see* (2.16) *p. 62 for the exact meaning of the parameters.).*
P+P2Level(m, n_V)	*Two level preconditioner where* $\widetilde{\mathrm{M}}^{-1}$ *=MSM with Sparse Patch P*(g, p, l_{max}, α) (2.16).

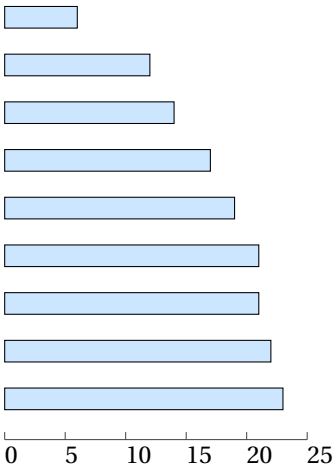
2D Case

In this experiment we test **scaled performance** (in terms of number of iterations) of two-level preconditioner combined with Sparse Patch Method (see §2.6 p. 60) The computational domains Ω is chosen to be the unit square Ω. We decomposed Ω into M^2 subdomains manually, keeping the resolution of each subdomains fixed i.e., the number of subdomains increases $\mathrm{M} = \{2,3,4,5,6,7,8,9,10\}$ while size of the subdomains stays constant ($n_x = n_y = 80$) i.e., the size of the global mesh increases with M^2.

Final results can be compare with reference solvers i.e., for a given number of parts (n_p) we present couple of results for different method given in form: *number of iteration (condition number).*

Results for 2D scaled performance test of two-level preconditioner combined with Sparse Patch Method

n_p	ASM+I(1)	P(1,2,3,1)	$\kappa_{\approx}$	n-iter	P+P2Level(7,3)
4	32 (160.00)	15 (24.00)	3.96	6	
9	42 (320.00)	24 (48.31)	10.43	12	
16	65 (546.26)	33 (82.18)	10.73	14	
25	73 (837.77)	39 (125.84)	12.89	17	
36	94 (1194.24)	47 (177.97)	16.51	19	
49	103 (1615.65)	52 (241.77)	17.16	21	
64	120 (2101.90)	61 (313.47)	17.40	21	
81	130 (3116.19)	93 (1402.03)	17.50	22	
100	144 (3890.81)	104 (1762.50)	18.00	23	



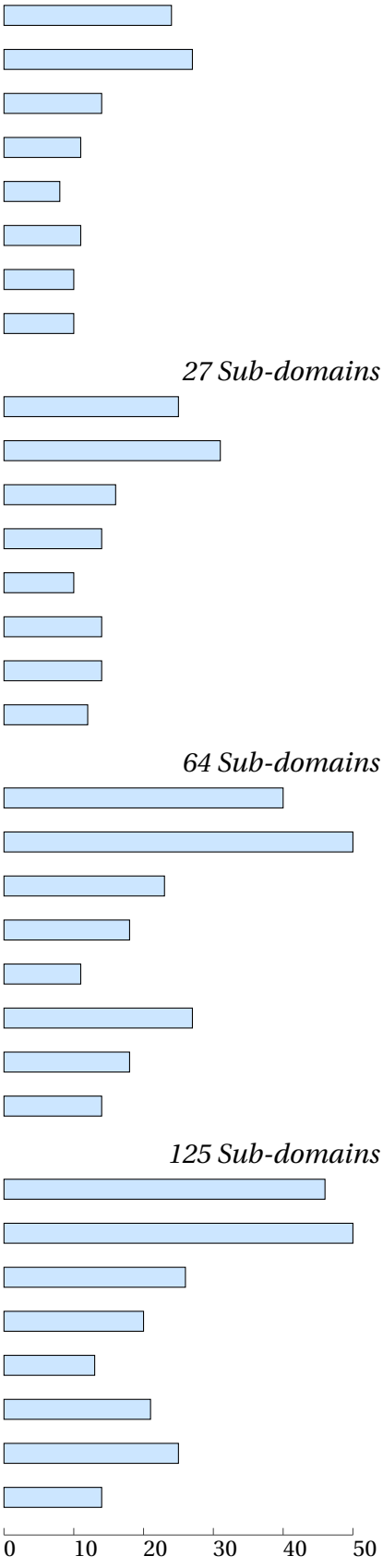
Comments 4.4.4. *This test shows how well the two-level preconditioner combined with Sparse Patch Method makes iterative solution insensitive (in terms of iterations) to both a growing number of sub-domain and more and more refined mesh. For example, from* $n_p = 49$ *to* $n_p = 100$ *we doubled the number of sub-domains but the number of iteration stay at the same level. In this case, combining the Sparse Patch method with our two-level preconditioner yields very good results.*

3D Case

In this numerical experiment Ω is the unit cube which we decomposed into M^3 sub-cubes, each of the same size. Hence, like in previous experiment we increase number of subdomain ($M = \{2,3,4,5\}$) keeping its size constant ($n_x = n_y = n_z = 20$).

Remark 4.5 (Final *error*)**.** *During our experiments we have noticed, that for a fixed tolerance (by default* $tol = 1e-6$*) for GMRES method used in all test, the final error*=$\max|[(\widetilde{F} - \widetilde{A}\widetilde{U}_{sol})]_i|$ *can strongly vary, which means that some methods brings some benefits also to quality of solution while keeping number of iterations relatively small. To visualise this phenomena we add an additional column to the results for the 3D case.*

Method	*error*	$\kappa_{\approx}$	*n-iter*	
M = 2				*8 Sub-domains*
ASM no I	1.756e-06	39.97	24	
ASM no I (*low tol*)	1.601e-07	39.97	27	
ASM + I(1)	2.558e-09	12.55	15	
ASM + I(2)	7.434e-09	6.84	11	
P2Level(5,3)	1.600e-08	3.32	8	
Patch(1,3,3,0)	8.498e-09	4.26	11	
Patch(2,3,3,0)	7.702e-10	3.56	10	
P(1,2,3,1)+P2Level(5,2)	1.007e-08	2.49	10	
M = 3				*27 Sub-domains*
ASM no I	1.743e-06	78.03	25	
ASM no I (*low tol*)	2.059e-08	78.03	31	
ASM + I(1)	1.093e-08	22.95	16	
ASM + I(2)	3.429e-09	12.71	14	
P2Level(5,3)	9.927e-08	4.93	10	
Patch(1,3,3,0)	1.010e-10	10.44	14	
Patch(2,3,3,0)	1.319e-09	7.03	14	
P(1,2,3,1)+P2Level(5,2)	1.433e-08	4.05	12	
M = 4				*64 Sub-domains*
ASM no I	2.307e-06	136.53	40	
ASM no I (*low tol*)	1.997e-08	136.53	50	
ASM + I(1)	6.937e-09	43.09	23	
ASM + I(2)	3.141e-09	24.07	18	
P2Level(5,3)	2.614e-08	5.85	11	
Patch(1,3,3,0)	1.534e-11	18.89	27	
Patch(2,3,3,0)	2.176e-10	12.52	18	
P(1,2,3,1)+P2Level(5,2)	3.161e-08	4.37	14	
M = 5				*125 Sub-domains*
ASM no I	1.810e-06	206.82	46	
ASM no I (*low tol*)	1.237e-08	206.03	55	
ASM + I(1)	9.315e-09	64.64	26	
ASM + I(2)	1.152e-08	36.00	20	
P2Level(5,3)	1.414e-07	7.98	13	
Patch(1,3,3,0)	2.337e-08	27.71	21	
Patch(2,3,3,0)	2.318e-12	19.65	25	
P(1,2,3,1)+P2Level(5,2)	4.354e-08	5.00	14	



Comments 4.4.5. *As in 2D case, the combined use of the Sparse Patch method along with ours two-level preconditioner yields iteration counts that are almost constant. Notice that in this case, the physical size of the number of subdomains increases.*

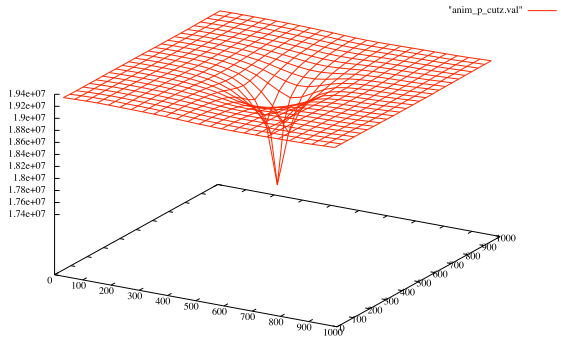
4.4.6 Reservoir simulations - experiment with Black Oil model

In order to test our new algebraic methods with linear systems which originate from porous media flow simulations, we simulated the five spot problem using well-known Blac-Oil model [15]. This model computes the saturation change of three phases (oil, water and gas) and pressure of each phase in each cell at each time step i.e, the Black-Oil model consists of a set of partial differential equations describing the conservation of mass for each component and the time evolution of the phase pressures and velocities. We limit our interest in this experiment to pressure block only.

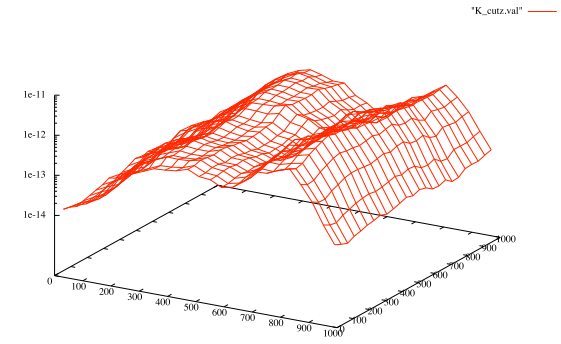
The variable which drives the condition number of our test matrices i.e., complexity of our experiment variant, is a permeability field of domain in which multi-phase flow is simulated. The reservoir permeability field is a full tensor quantity that presents very large local variations, up to 4 or 10 orders of magnitude [21]. This results in highly discontinuous terms in the discretized form of the equation that may lead to inaccurate solutions (example of permeability field used in our experiment is depicted on figure 4.2(b)). According to the value of these jumps we have for given size three matrices with suffices **var 4**, **var 8** and **var 12**. The higher is the number, the higher the jumps.

The five-spot problem consists of four injection wells symmetrically disposed around one production well (see figure 4.2(c)). More precisely, our computation domain is chosen to be parallelepiped Ω in size $(0,1000) \times (0,1000) \times (0,100) m$. Production well is placed in the middle of Ω ($x = y = 500m$) and it perforates each cell in $(L_z/2, L_z)$, where L_z is length in z direction. The average permeability K is considered to be isotropic in x and y direction ($K_x = K_y$) and highly anisotropic in z, therefore in order to create sub-domains, we will make "cuts" only on plane xy or we use graph partitioner (without weights).

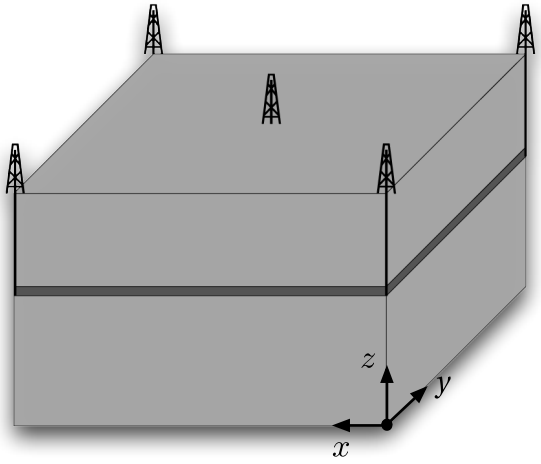
Two sizes of a uniform mesh are used for discretizing the Black-Oil model on Ω; $30 \times 30 \times 16$ and $60 \times 60 \times 32$. In order to distinguish which type of mesh and permeability field was used in experiment we simply denote each variant by: *$(x \times y \times z)$var(4,8 or 12)*.



(a) Pressure final solution on XY plane for Z/2.



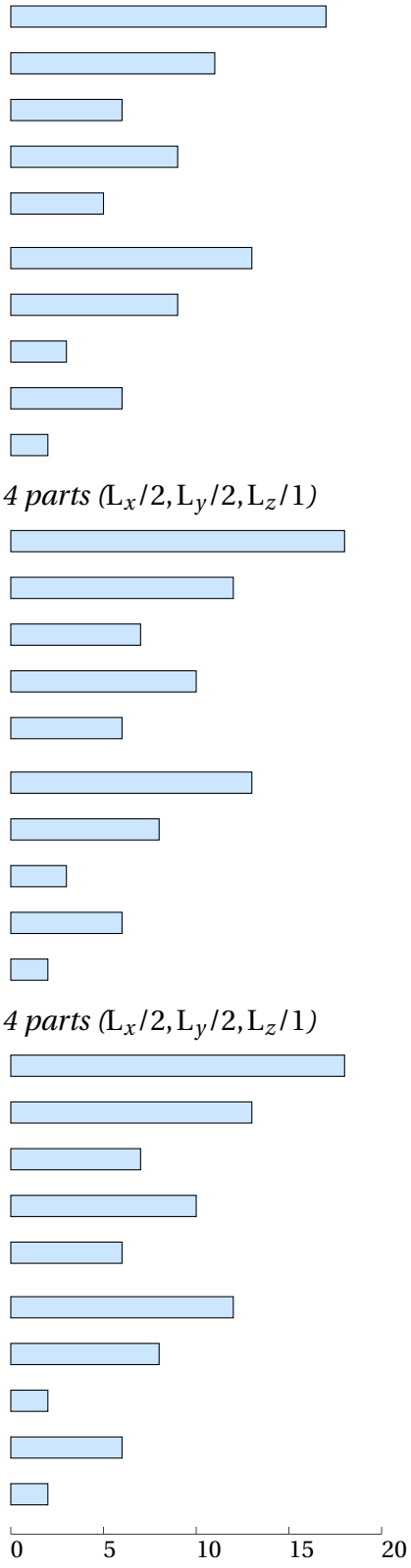
(b) Permeability values on XY plane for Z/2 (field example for var 4).



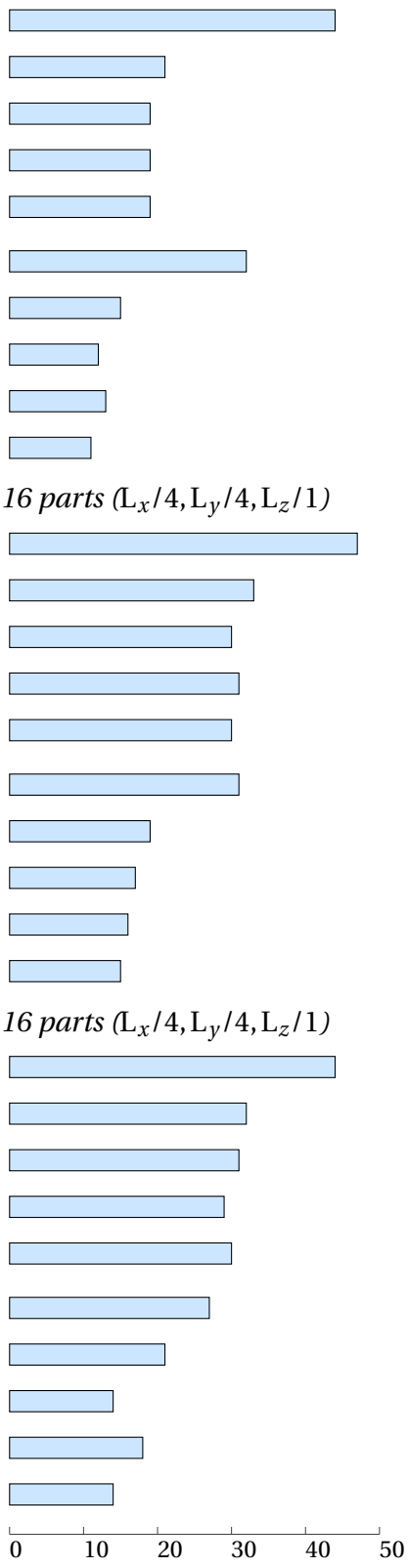
(c) Configuration of wells in five-spot problem

Figure 4.2: Five-spot problem for Blac-Oil model simulation.

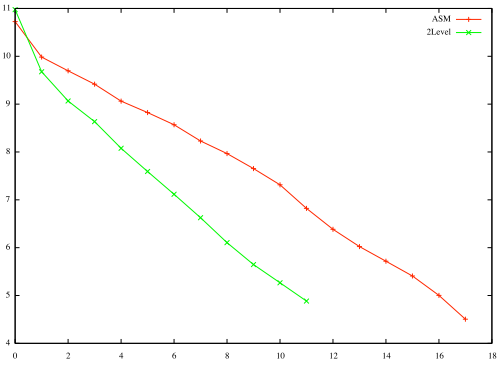
Method	$\kappa_{\approx}$	*n-iter*	
30x30x16 var 4			*4 parts* $(L_x/2, L_y/2, L_z/1)$
Ref. ASM + Inf(1)	10.13	17	
Successive P2Level(10,2)	3.74	11	
Adaptive P2Level(10,2)	3.80	6	
Successive P2Level(10,4)	3.04	9	
Adaptive P2Level(10,4)	2.96	5	
Ref. ASM + Inf(2)	5.59	13	
Successive P2Level(10,2)	2.67	9	
Adaptive P2Level(10,2)	1.82	3	
Successive P2Level(10,4)	1.31	6	
Adaptive P2Level(10,4)	1.25	2	
30x30x16 var 8			*4 parts* $(L_x/2, L_y/2, L_z/1)$
Ref. ASM + Inf(1)	12.79	18	
Successive P2Level(10,2)	6.27	12	
Adaptive P2Level(10,2)	6.00	7	
Successive P2Level(10,4)	5.97	10	
Adaptive P2Level(10,4)	4.04	6	
Ref. ASM + Inf(2)	5.97	13	
Successive P2Level(10,2)	2.45	8	
Adaptive P2Level(10,2)	2.00	3	
Successive P2Level(10,4)	1.72	6	
Adaptive P2Level(10,4)	1.00	2	
30x30x16 var 12			*4 parts* $(L_x/2, L_y/2, L_z/1)$
Ref. ASM + Inf(1)	17.00	18	
Successive P2Level(10,2)	9.75	13	
Adaptive P2Level(10,2)	7.77	7	
Successive P2Level(10,4)	8.86	10	
Adaptive P2Level(10,4)	3.96	6	
Ref. ASM + Inf(2)	5.80	12	
Successive P2Level(10,2)	2.62	8	
Adaptive P2Level(10,2)	1.39	2	
Successive P2Level(10,4)	1.64	6	
Adaptive P2Level(10,4)	1.22	2	



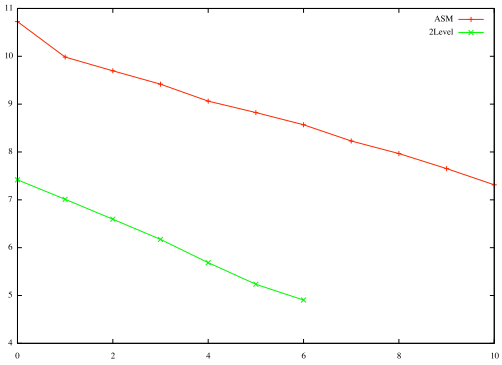
Method	$\kappa_{\approx}$	*n-iter*	
60x60x32 var 4			*16 parts* $(L_x/4, L_y/4, L_z/1)$
Ref. ASM + Inf(1)	81.28	44	
Successive P2Level(10,2)	15.36	21	
Adaptive P2Level(10,2)	15.24	19	
Successive P2Level(10,4)	15.95	19	
Adaptive P2Level(10,4)	16.09	19	
Ref. ASM + Inf(2)	37.72	32	
Successive P2Level(10,2)	7.76	15	
Adaptive P2Level(10,2)	7.70	12	
Successive P2Level(10,4)	6.60	13	
Adaptive P2Level(10,4)	6.64	11	
60x60x32 var 8			*16 parts* $(L_x/4, L_y/4, L_z/1)$
Ref. ASM + Inf(1)	123.70	47	
Successive P2Level(10,2)	59.07	33	
Adaptive P2Level(10,2)	58.91	30	
Successive P2Level(10,4)	58.69	31	
Adaptive P2Level(10,4)	58.59	30	
Ref. ASM + Inf(2)	48.90	31	
Successive P2Level(10,2)	14.49	19	
Adaptive P2Level(10,2)	14.50	17	
Successive P2Level(10,4)	12.19	16	
Adaptive P2Level(10,4)	12.55	15	
60x60x32 var 12			*16 parts* $(L_x/4, L_y/4, L_z/1)$
Ref. ASM + Inf(1)	108.04	44	
Successive P2Level(10,2)	86.91	32	
Adaptive P2Level(10,2)	85.48	31	
Successive P2Level(10,4)	82.96	29	
Adaptive P2Level(10,4)	84.73	30	
Ref. ASM + Inf(2)	31.71	27	
Successive P2Level(10,2)	25.54	21	
Adaptive P2Level(10,2)	23.42	14	
Successive P2Level(10,4)	23.53	18	
Adaptive P2Level(10,4)	21.86	14	



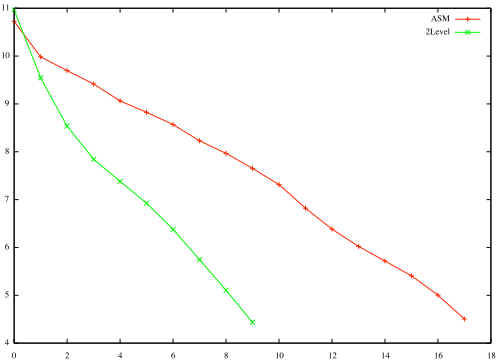
Matrix variant: $\mathbf{30 \times 30 \times 16\ var\ 4 + Inf(1)}$



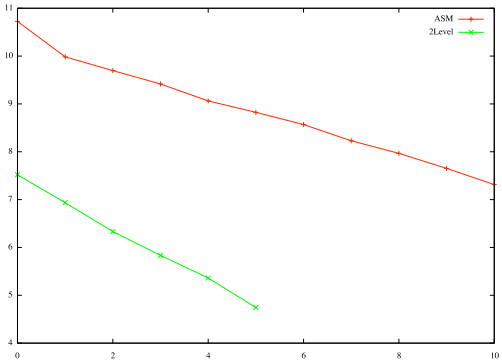
Successive P2Level(10,2)



Adaptive P2Level(10,2)

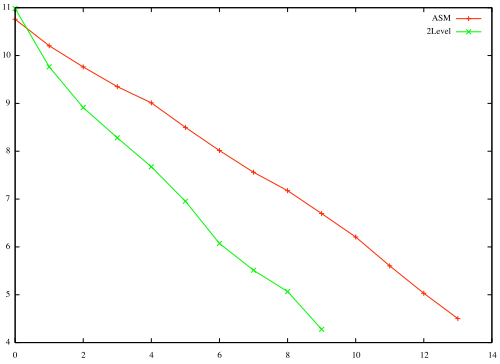


Successive P2Level(10,4)

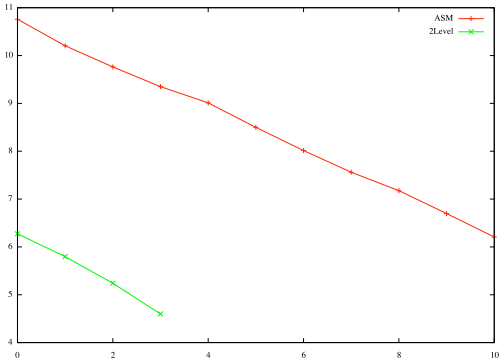


Adaptive P2Level(10,4)

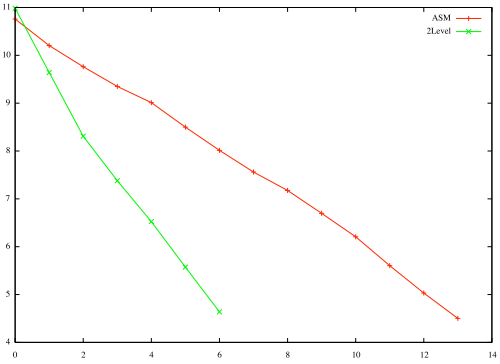
Matrix variant: $\mathbf{30 \times 30 \times 16\ var\ 4 + Inf(2)}$



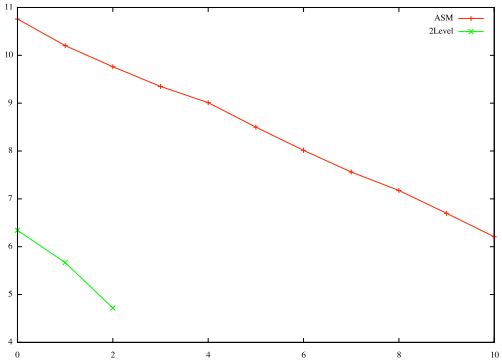
Successive P2Level(10,2)



Adaptive P2Level(10,2)

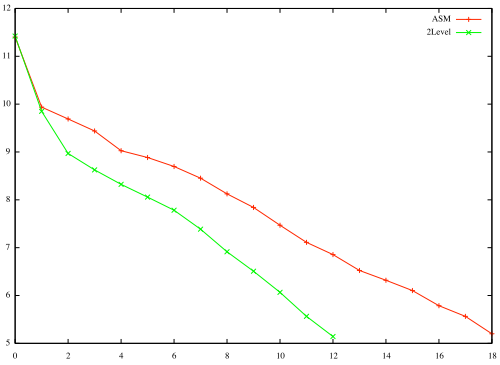


Successive P2Level(10,4)

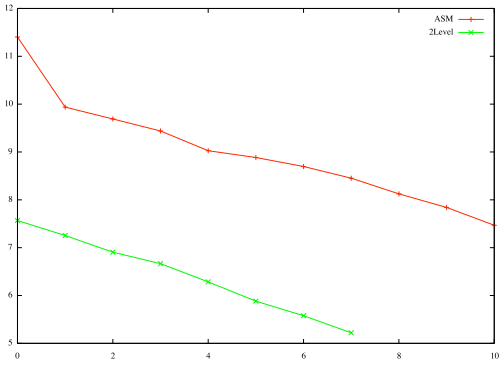


Adaptive P2Level(10,4)

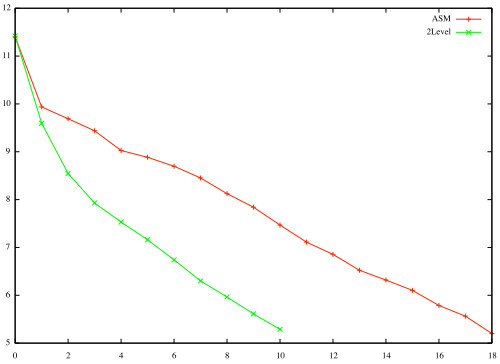
Matrix variant: **30 × 30 × 16 var 8 + Inf(1)**



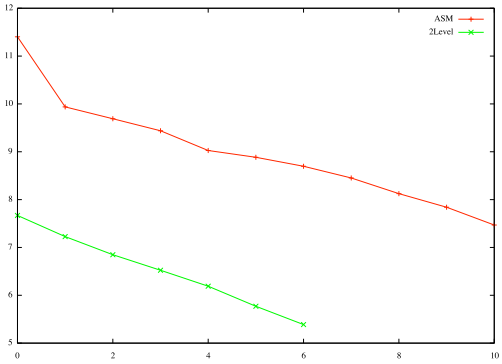
Successive P2Level(10,2)



Adaptive P2Level(10,2)

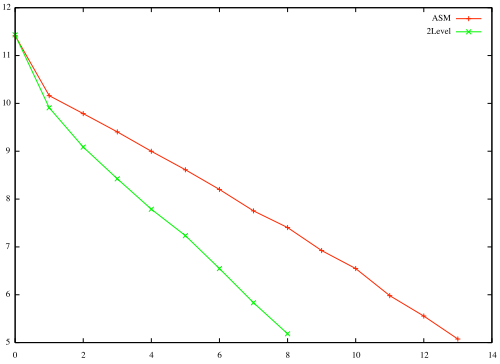


Successive P2Level(10,4)

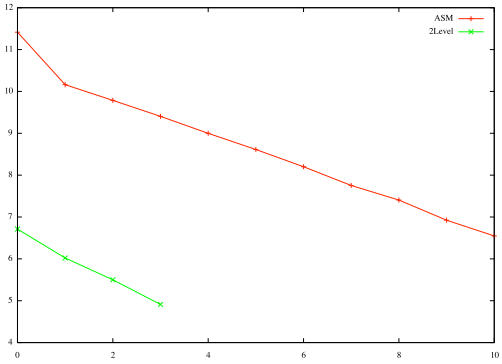


Adaptive P2Level(10,4)

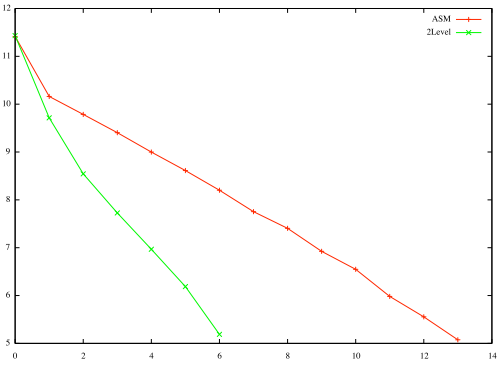
Matrix variant: **30 × 30 × 16 var 8 + Inf(2)**



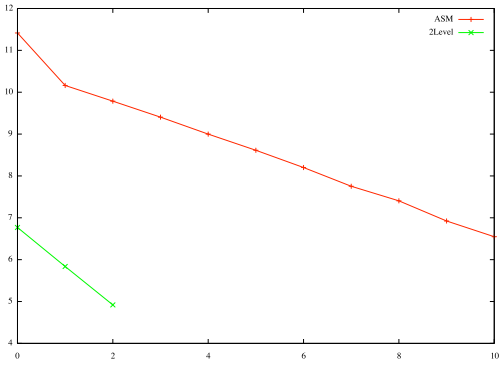
Successive P2Level(10,2)



Adaptive P2Level(10,2)

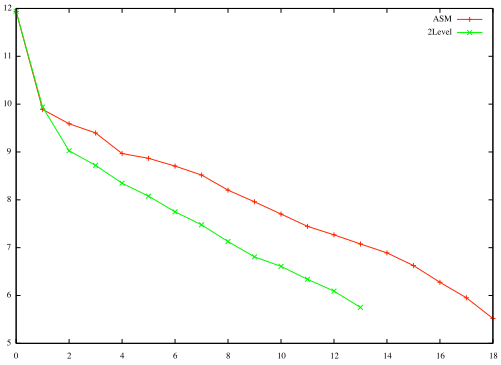


Successive P2Level(10,4)

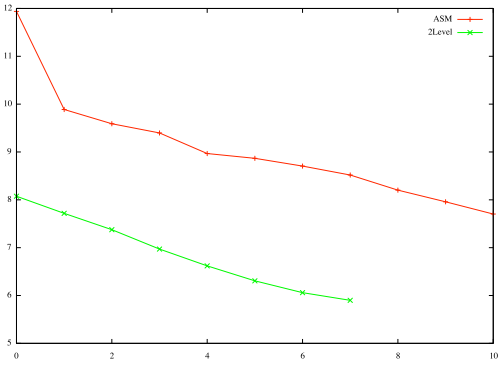


Adaptive P2Level(10,4)

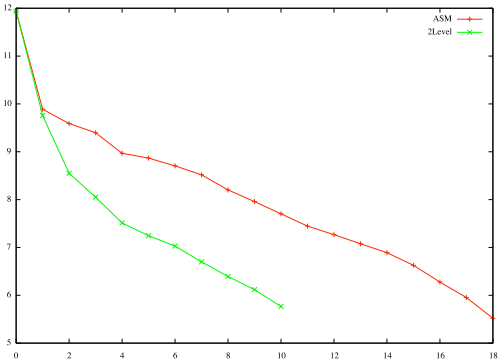
Matrix variant: **30 × 30 × 16 var 12 + Inf(1)**



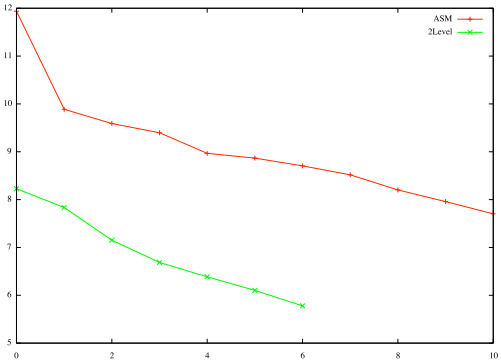
Successive P2Level(10,2)



Adaptive P2Level(10,2)

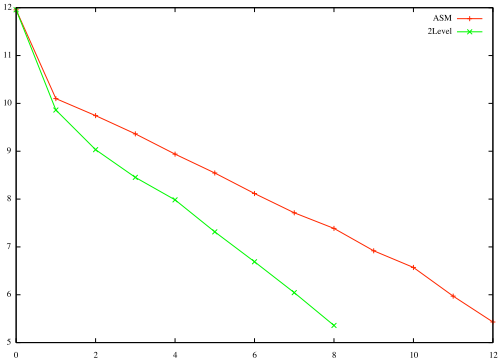


Successive P2Level(10,4)

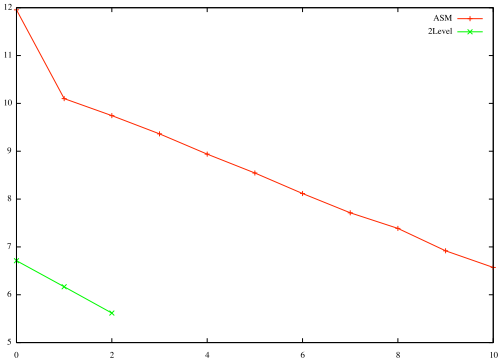


Adaptive P2Level(10,4)

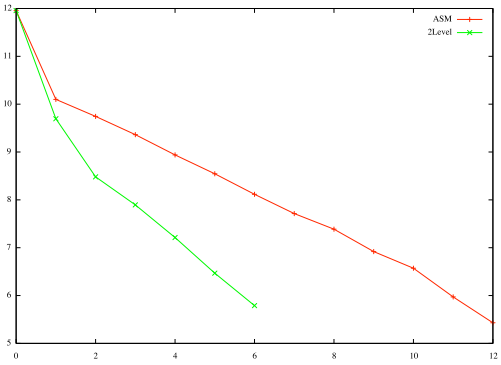
Matrix variant: **30 × 30 × 16 var 12 + Inf(2)**



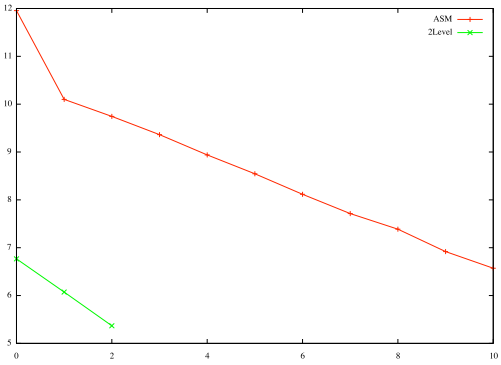
Successive P2Level(10,2)



Adaptive P2Level(10,2)

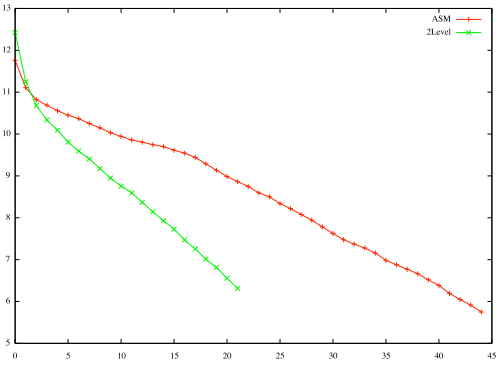


Successive P2Level(10,4)

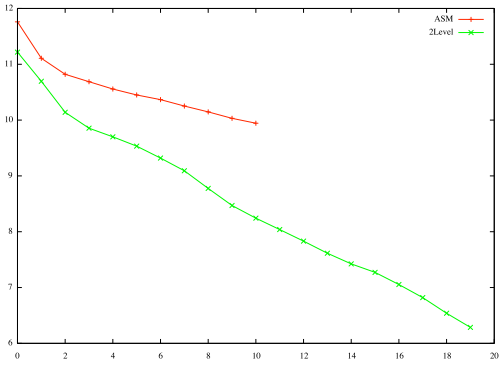


Adaptive P2Level(10,4)

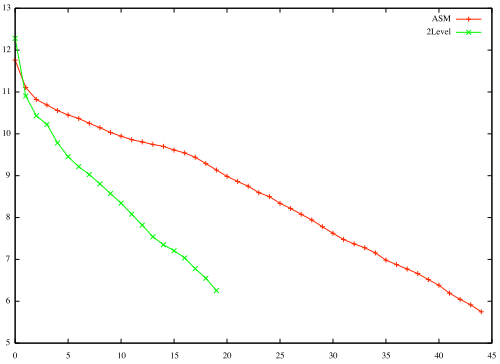
Matrix variant: **60 × 60 × 32 var 4 + Inf(1)**



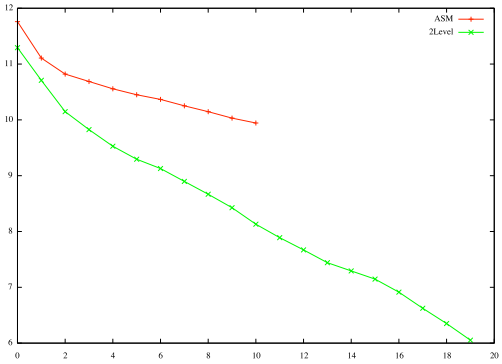
Successive P2Level(10,2)



Adaptive P2Level(10,2)

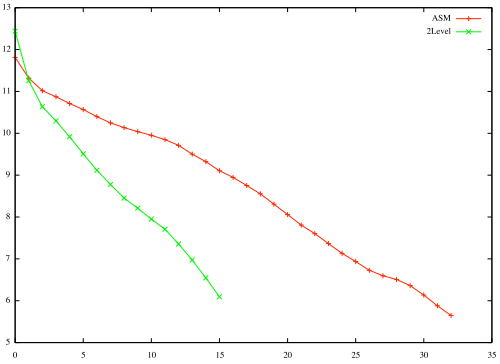


Successive P2Level(10,4)

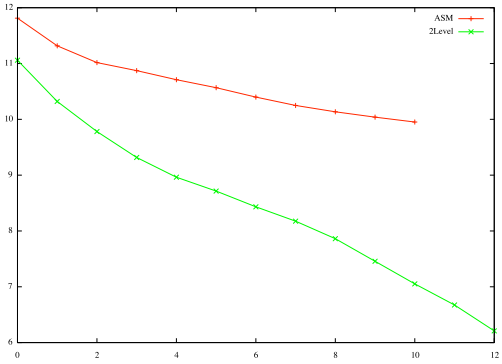


Adaptive P2Level(10,4)

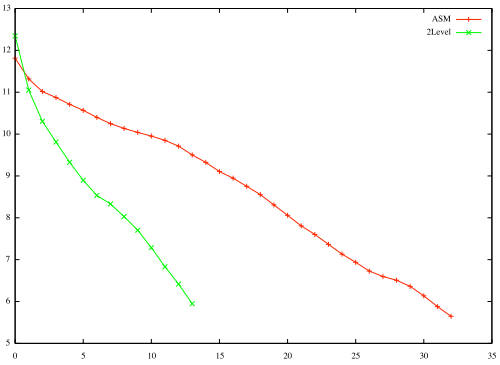
Matrix variant: **60 × 60 × 32 var 4 + Inf(2)**



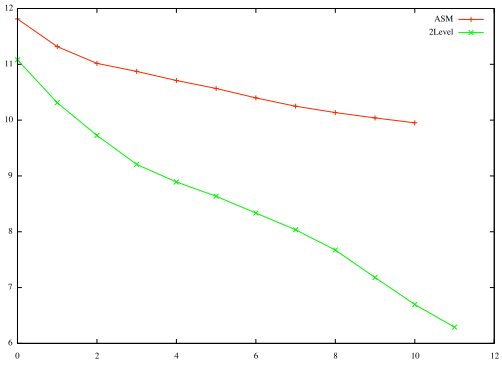
Successive P2Level(10,2)



Adaptive P2Level(10,2)

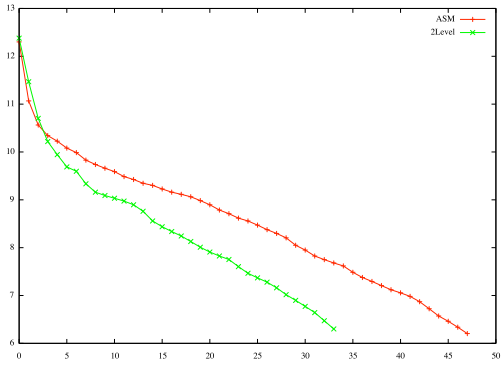


Successive P2Level(10,4)

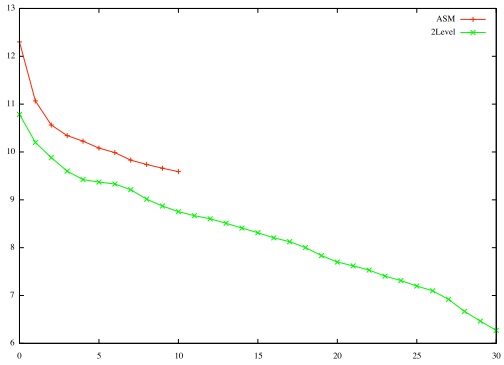


Adaptive P2Level(10,4)

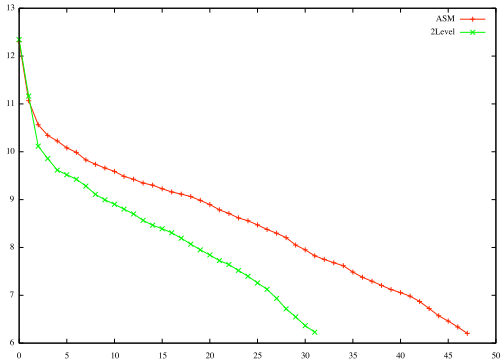
Matrix variant: **60 × 60 × 32 var 8 + Inf(1)**



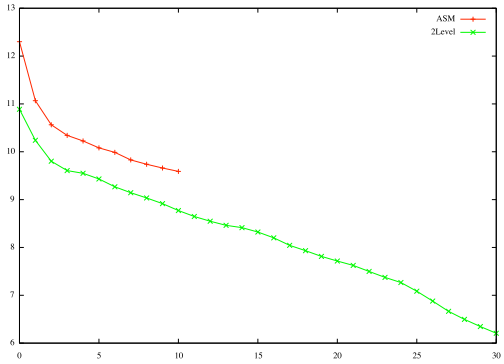
Successive P2Level(10,2)



Adaptive P2Level(10,2)

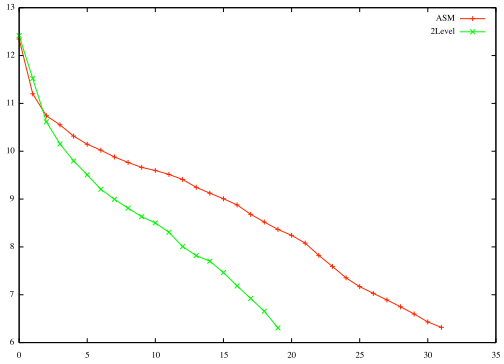


Successive P2Level(10,4)

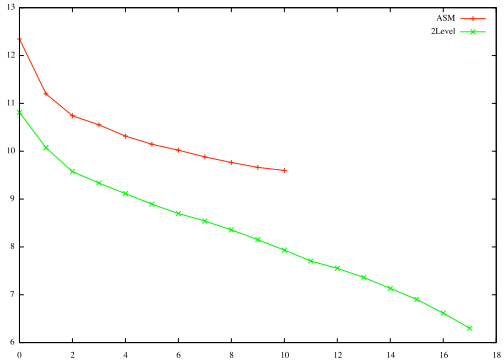


Adaptive P2Level(10,4)

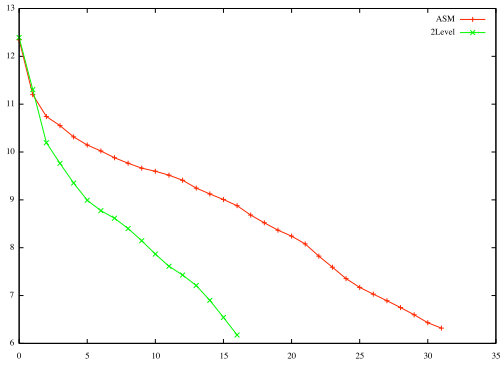
Matrix variant: **60 × 60 × 32 var 8 + Inf(2)**



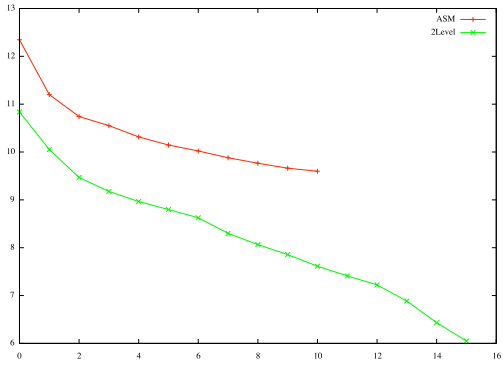
Successive P2Level(10,2)



Adaptive P2Level(10,2)

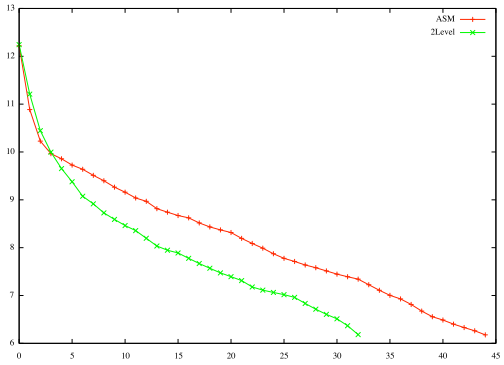


Successive P2Level(10,4)

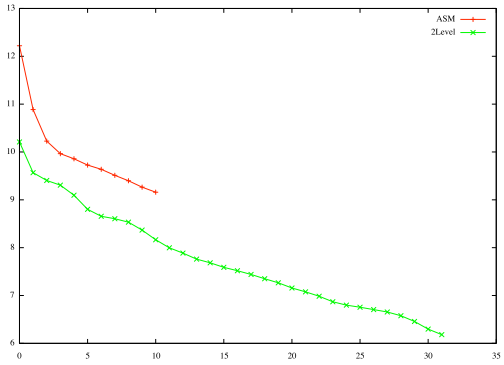


Adaptive P2Level(10,4)

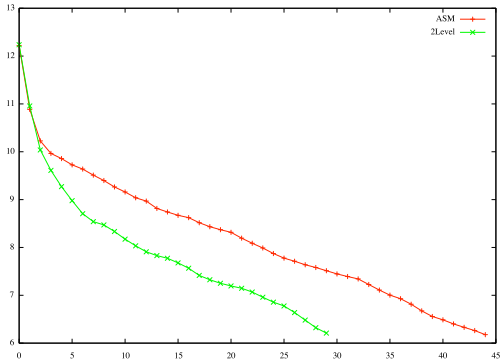
Matrix variant: **60 × 60 × 32 var 12 + Inf(1)**



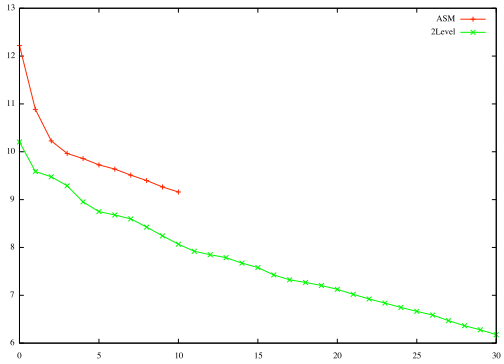
Successive P2Level(10,2)



Adaptive P2Level(10,2)

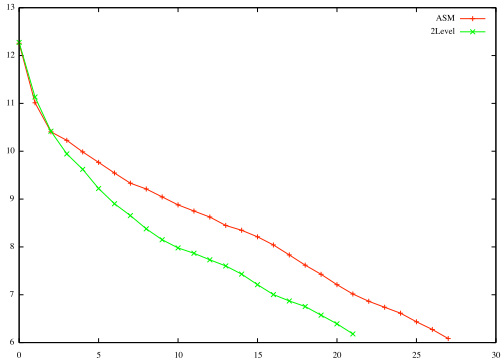


Successive P2Level(10,4)

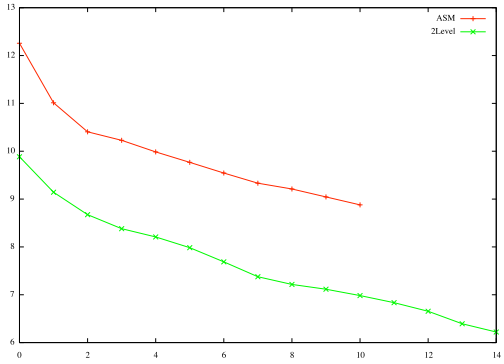


Adaptive P2Level(10,4)

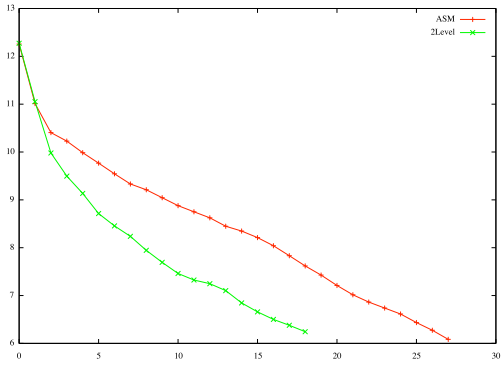
Matrix variant: **60 × 60 × 32 var 12 + Inf(2)**



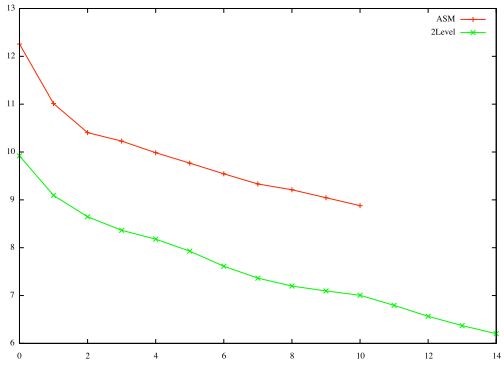
Successive P2Level(10,2)



Adaptive P2Level(10,2)

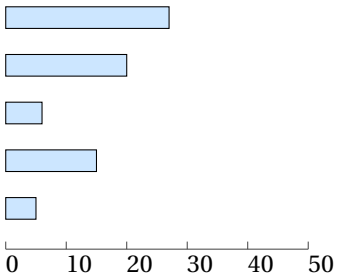


Successive P2Level(10,4)

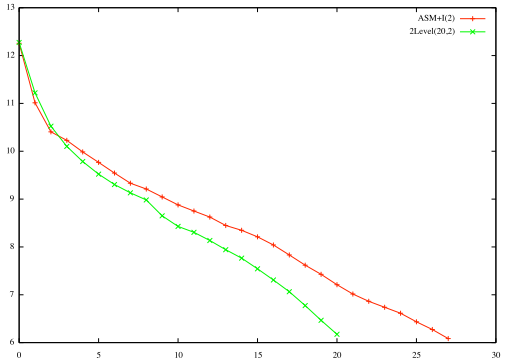


Adaptive P2Level(10,4)

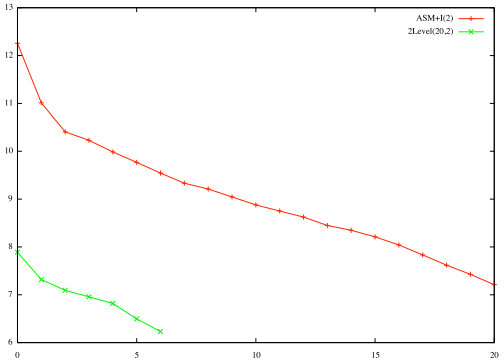
Method	$\kappa_{\approx}$	*n-iter*
Ref. ASM + Inf(2)	31.71	27
Successive P2Level(20,2)	19.25	20
Adaptive P2Level(20,2)	10.59	6
Successive P2Level(20,4)	11.33	15
Adaptive P2Level(20,4)	7.50	5



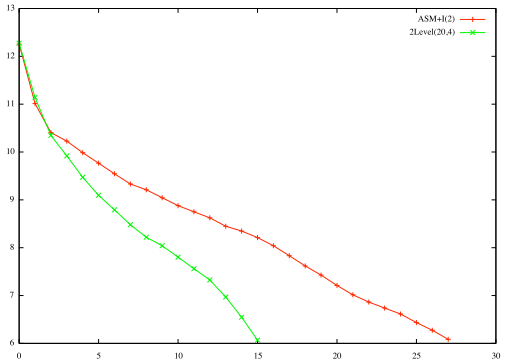
Matrix variant: **60 × 60 × 32 var 12**



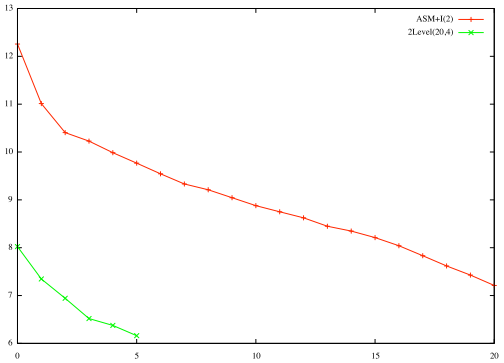
Successive P2Level(20,2)



Adaptive P2Level(20,2)



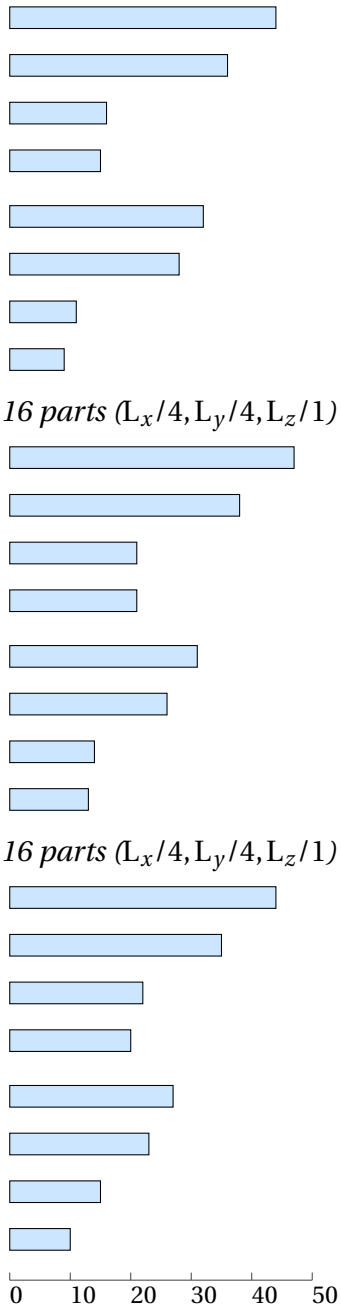
Successive P2Level(20,4)



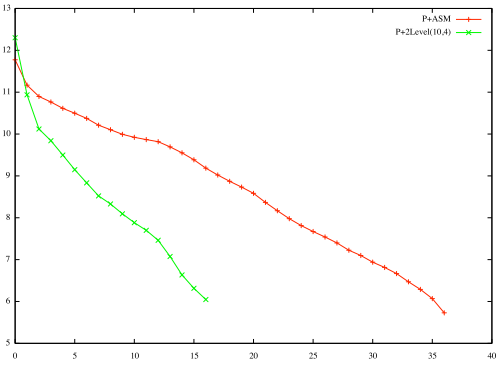
Adaptive P2Level(20,4)

In following experiment we build coarse space using approximated eigenvectors from Patched operator i.e., we use Sparse Patch method for first solve.

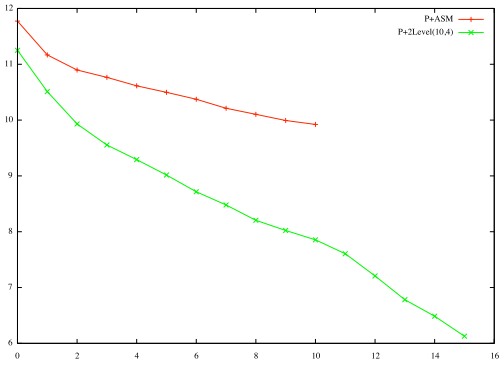
Method	$\kappa_{\approx}$	*n-iter*	
60x60x32 var 4			*16 parts* $(L_x/4, L_y/4, L_z/1)$
Ref. ASM + Inf(1)	81.28	44	
Ref. ASM + Patch(1,2,3,1)	56.16	36	
Successive P+P2Level(10,4)	11.47	16	
Adaptive P+P2Level(10,4)	11.51	15	
Ref. ASM + Inf(2)	37.72	32	
Ref. ASM + Patch(2,2,3,1)	29.52	28	
Successive P+P2Level(10,4)	4.85	11	
Adaptive P+P2Level(10,4)	4.89	9	
60x60x32 var 8			*16 parts* $(L_x/4, L_y/4, L_z/1)$
Ref. ASM + Inf(1)	123.70	47	
Ref. ASM + Patch(1,2,3,1)	78.56	38	
Successive P+P2Level(10,4)	32.86	21	
Adaptive P+P2Level(10,4)	32.91	21	
Ref. ASM + Inf(2)	48.90	31	
Ref. ASM + Patch(2,2,3,1)	36.79	26	
Successive P+P2Level(10,4)	8.41	14	
Adaptive P+P2Level(10,4)	8.50	13	
60x60x32 var 12			*16 parts* $(L_x/4, L_y/4, L_z/1)$
Ref. ASM + Inf(1)	108.04	44	
Ref. ASM + Patch(1,2,3,1)	66.69	35	
Successive P+P2Level(10,4)	48.79	22	
Adaptive P+P2Level(10,4)	48.10	20	
Ref. ASM + Inf(2)	31.71	27	
Ref. ASM + Patch(2,2,3,1)	23.55	23	
Successive P+P2Level(10,4)	16.74	15	
Adaptive P+P2Level(10,4)	16.84	10	



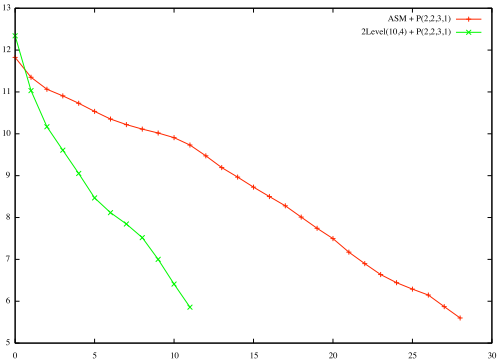
Matrix variant: **60 × 60 × 32 var 4**



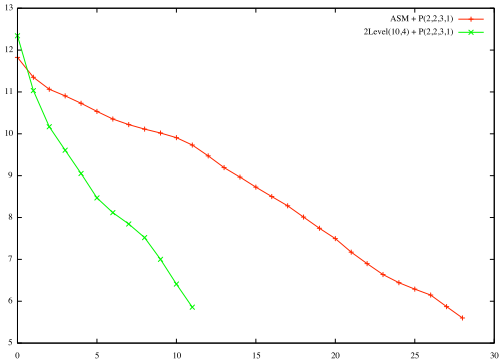
Successive P2Level(10,4)+P(1,2,3,1)



Adaptive P2Level(10,4)+P(1,2,3,1)

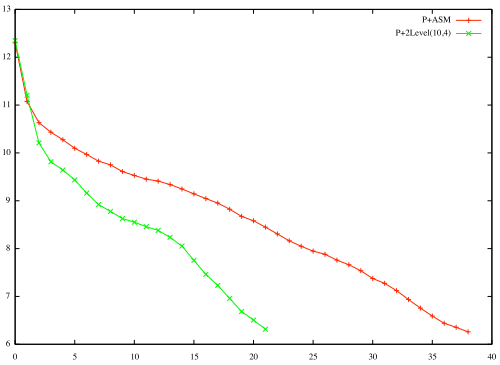


Successive P2Level(10,4)+P(2,2,3,1)

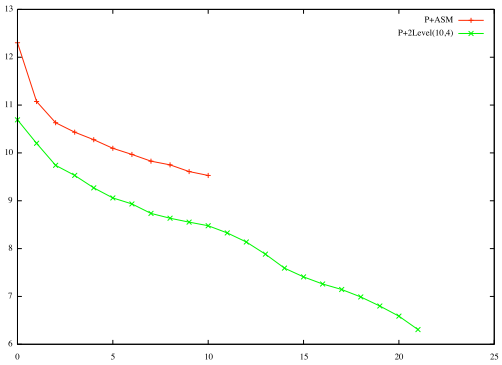


Adaptive P2Level(10,4)+P(2,2,3,1)

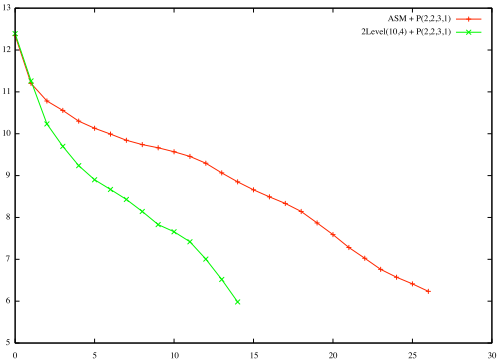
Matrix variant: **60 × 60 × 32 var 8**



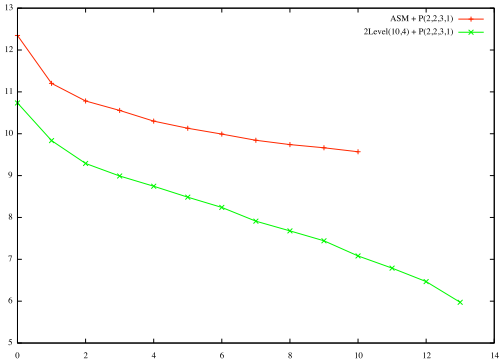
Successive P2Level(10,4)+P(1,2,3,1)



Adaptive P2Level(10,4)+P(1,2,3,1)

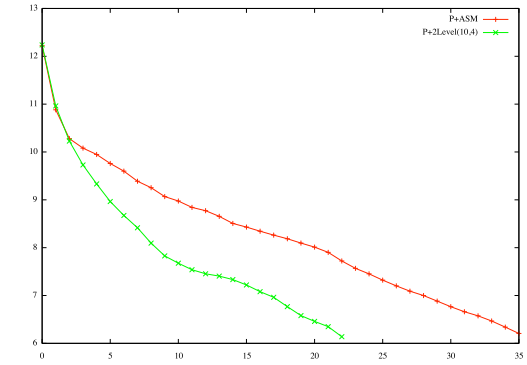


Successive P2Level(10,4)+P(2,2,3,1)

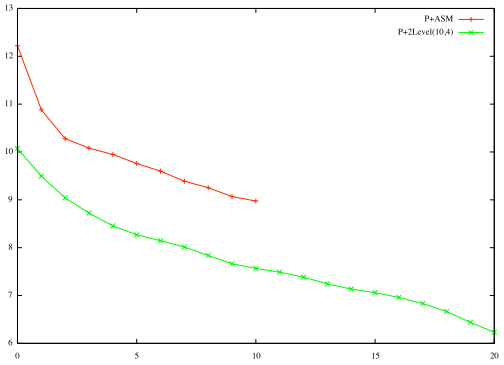


Adaptive P2Level(10,4)+P(2,2,3,1)

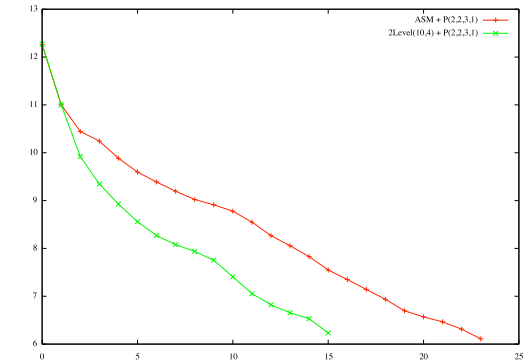
Matrix variant: **60 × 60 × 32 var 12**



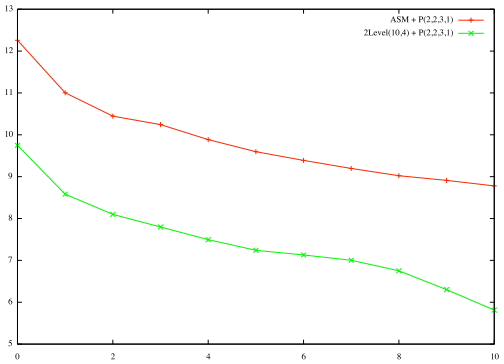
Successive P2Level(10,4)+P(1,2,3,1)



Adaptive P2Level(10,4)+P(1,2,3,1)



Successive P2Level(10,4)+P(2,2,3,1)



Adaptive P2Level(10,4)+P(2,2,3,1)

Comments 4.4.6. *We see from the plots that the size of the overlap is very important in these cases and the Schwarz method is very robust to the variance of the permeability field. As for the two level method, for matrices **var 4** and **var 8**, setting $m = 10$ is enough to significantly change the slope of the convergence curve as compared to the one-level method. But for the highest (which is rather unrealistic case) variance **var 12**, we had to fix $m = 20$ in order to improve over the one-level method.*

Chapter 5

Numerical Experiments

In contrary to subsections *"Numerical results"* at the end of previous chapters, numerical results present in this chapter intend to present how ADDMlib cope with real test cases and academical problems in "industrial" [1] size and difficulties.

5.1 3D Laplace problem

Numerical Experiment 5.1. *Let us consider Laplace equation* $-\Delta(\mathbf{u}) = 0$*, discretized using P13D-type finite elements on hexahedral domain* Ω *of a size:* $m\frac{nx}{4} \times m\frac{ny}{4} \times m\frac{nz}{4}$ *with Dirichlet condition on the top face (*$\mathbf{u} = 1.0$ *for* $z = \frac{nz}{4}$ *and* $nz \in \{10, 15\}$*) and homogeneous Neumann conditions on the other boundaries.*

5.1.1 Setup of the Experiment 5.1

A finite elements discretization scheme was applied to the set of discretized problems, where we use tetrahedral mesh. Discretization and mesh construction has been performed via FreeFem++ [34]. For the domain decomposition into N subdomains we performed manual partition in such a way that subdomains shape cubes of the same size $[\Omega_n]_{\texttt{SIZE}} = m^3$. The righthand side of the resulting linear system is a function f which gives random values from set $\langle 1, 2)$ (computed once and before an experiment). We solved the linear system using GMRES method preconditioned by a preconditioner specified by variant of experiment (one level Additive Schwarz Method or two-level preconditioner). For the subdomain $(\mathrm{A}_{\widetilde{\Omega}_i}^{-1})$ and coarse operator (E^{-1}) solvers we chose the direct solver *SuperLU* [18]. The initial guess is chosen to be $\mathbf{u}_0 = 0$ and the stopping criterion $\|r_i\| \leq tol \cdot \|r_0\|$ for default $tol = 1 \times 10^{-6}$ and r_i to be a residual. The roughly estimated condition number (see remark 5.1 p. 118) is given as $\kappa_{\approx} = \lambda_{max}/\lambda_{min}$ where $\lambda_{\{min,max\}}$ are the approximated, extreme eigenvalues of $(\widetilde{\mathrm{M}}^{-1}\widetilde{\mathrm{A}})$ or $(\widetilde{\mathrm{M}}_{\star}^{-1}\widetilde{\mathrm{A}}_{\star})$.

Since our two-level method (see §4) need a set of the approximated eigenvectors in order to construct coarse space we executed each numerical test (except case with no inflation) as

1. Up to 3.5×10^6 of unknowns.

a concatenation of two iterative processes. A "classical", one-level Additive Schwarz Method with a different size of overlap (inflation depth) and the iterative method with two-level preconditioner built from "informations" collected during first solve (approximated eigenvectors). For this reason we divided our experiment into number of stages.

In first stage we read sparse matrix from a file. The matrix is next partitioned and distributed in order to define `DDMOperator`. Resulting operator is inflated (according to set up level) and then it is solved by the classical Schwarz method (which involves LU decomposition of each endomorphic `Partial Operator`). At the end of the first solve, the spectrum of upper Hessenberg matrix is automatically analyse and for each eigenvalue which $Re(\lambda_i) < tol = 0.1$, the approximated eigenvector of the preconditioned system is compute (see §3.2). If in the analysing spectrum there is no eigenvalues satisfying given threshold, only one eigenvector corresponding to the smallest eigenvalue is compute. On the other hand if spectrum consist a big number of small eigenvalues we sometimes limit our interest to the N_v smallest values, which implies that for a coarse space we use N_v approximated eigenvectors (in tables and plots this variant is denoted by "VF"). The resulting collection of eigenvectors is use to create the coarse space operator (see §4.3) which is next assemble with "Schwarz method" preconditioner (builded for the first solve) in order to create two-level preconditioner. Finally we solve our linear system once again with the new preconditioner, zero initial guess and the same level of inflation i.e., we reuse `DDMOperator` from the first solve.

Figure 5.1: Stages of experiment 5.1 on time axis.

The results for each solve (experiment variants) are presented in two ways. First, results will be summarised in a table, presenting number of iterations, roughly approximated condition number, standard norm of the relative errors (i.e., $||Au_{it} - b||_2 / ||b||_2$ [2] with the iterated solution u_{it}) and number of approximated eigenvectors used in construction of Coarse Operator E. In addition we also included average time (in seconds) of execution major stages during our experiment. Second, the results are represented graphically, by showing relative error of residual in 2-norm during the iteration process. On top of each set of results (table and plot) the reader will find essential information about solved matrix (number of rows and non zero values in it) and domain which we use to generate linear system (e.g., size of subdomain).

2. For the system in form: $Au = b$.

High Performance tests

All numerical experiments in this chapter have been performed on a super computer. Numerical experiments with 3D Laplace have been performed on *IFP Energies Nouvelles'* cluster of 114 nodes equipped with 4 processes AMD Barcelona 2.3 Ghz (each with quad-core socket) interconnected by Infiniband switched fabric (type of network topology). Maximum number of available processes was limited to 256, therefore in variants of experiment in which computational domain was decomposed into more then 256 sub-domains, we dedicated more then one sub-domain per one process. Otherwise each sub-domain (ADDMlib `Part`) was dedicated to one process.

Remaining numerical experiments we performed on UMPC cluster, which is a 80 nodes of 2 quad-core processes Xeon Nehalem 2.53Ghz also interconnected by Infiniband switched fabric. Maximum number of available processes was limited to 80.

Notation 5.1. *For the each different size of computational domain, the reader will find a result of different variants of numerical experiment collected in table with following notation:*

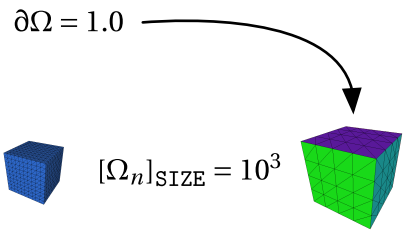
expvar	*solver variant where `I(x)` denote Additive Schwarz Method with x depth of inflation and `D..` (or `(DEFLATION)`) is a version of experiment in which GMRES method was preconditioned by two-level preconditioner.*
niter	*number of iterations*
κ	*roughly estimated condition number given as* $\kappa = \lambda_{max}/\lambda_{min}$ *where* $\lambda_{\{min,max\}}$ *are the approximated, extreme eigenvalues of* $(\widetilde{\mathrm{M}}^{-1}\widetilde{\mathrm{A}})$ *or* $(\widetilde{\mathrm{M}}_{\star}^{-1}\widetilde{\mathrm{A}}_{\star})$
nV	*number of approximated eigenvectors used in construction of coarse space*
$\|\|r_{sol}\|\|$	*standard norm of final residual i.e.,* $\|\|r_{sol}\|\| = \|\|\mathrm{A}u_{sol} - b\|\|_2/\|\|b\|\|_2$
CS$[s]$	*time of "construction" coarse space operator*
Inf$[s]$	*time of inflation process for each level*
LU$[s]$	*time of LU factorisation of endomorphic `Partial Operators` in `DDMOperator`*
sol$[s]$	*time of iterative process (in case of varian with two-level preconditioner* ***sol*** *consist also LU factorisation time of coarse operator)*

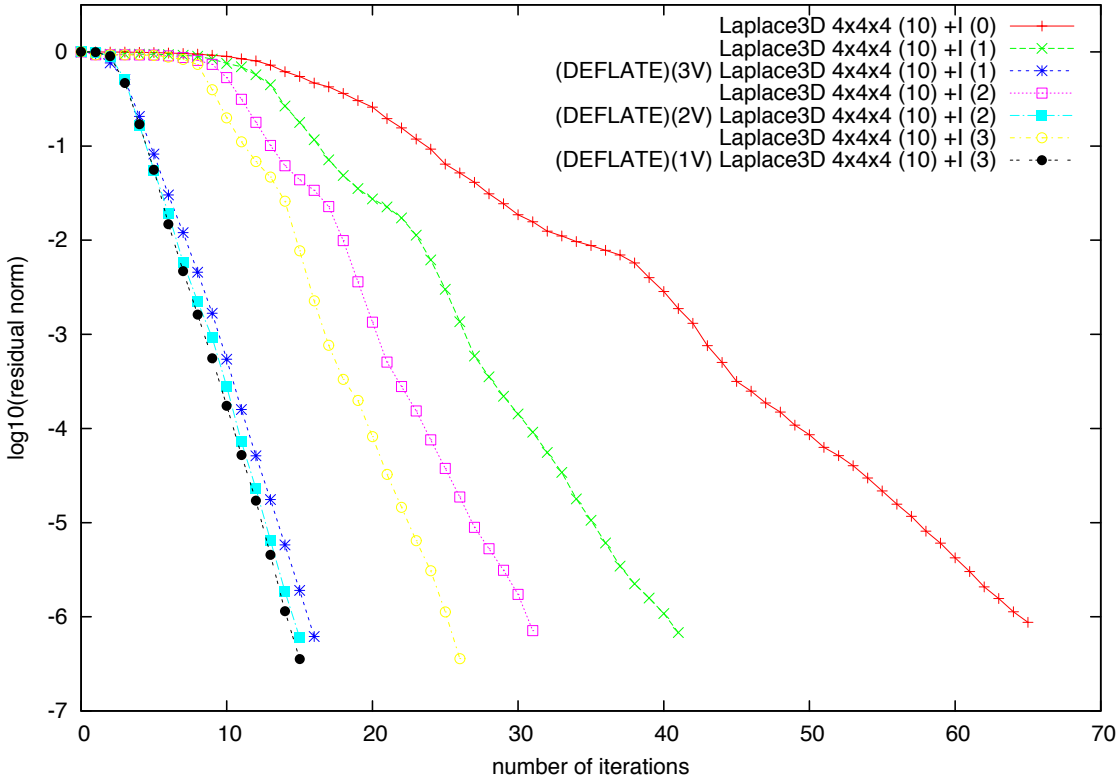
Remark 5.1. *In the field of numerical analysis, the condition number (usually denoted by* κ*) is a measure of sensitivity of a matrix (or the linear system it represents) to numerical operations. The condition number for the matrix inversion with respect to a matrix norm* $\|\cdot\|$ *of a square matrix* A *is defined by* $\kappa(\mathrm{A}) = \|\mathrm{A}\|\,\|\mathrm{A}^{-1}\|$*, if* A *is non-singlular; and* $\kappa \to +\infty$ *if* A *is singular. Of course, this definition depends on the choice of the norm i.e., if* $\|\cdot\|$ *is a* l_2 *and* A *is normal then* $\kappa\mathrm{A} = |\lambda_{max}(\mathrm{A})/\lambda_{min}(\mathrm{A})|$ *where* $\lambda_{max}(\mathrm{A})$ *and* $\lambda_{min}(\mathrm{A})$ *are maximal and minimal (by moduli) eigenvalues of A respectively.*

Since in our methods we use approximated eigenvalues extracted from Krylov space, we can define a similar *number using the same definition. Therefore for most of the experiment we compute* $\kappa_{\approx} = |\lambda^{\approx}_{max}(\mathrm{M}^{-1}\mathrm{A})/\lambda^{\approx}_{min}(\mathrm{M}^{-1}\mathrm{A})|$*, where* $\lambda^{\approx}_{max}(\mathrm{M}^{-1}\mathrm{A})$ *and* $\lambda^{\approx}_{min}(\mathrm{M}^{-1}\mathrm{A})$ *are approximated maximal and minimal eigenvalues of the operator* $\mathrm{M}^{-1}\mathrm{A}$*, where* M^{-1} *is a preconditioner use in iterative method.*

Matrix L3D4x4x4n10.mtx

nparts	nrows	nnz
64	68,921	993,961



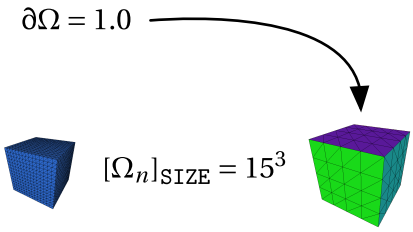


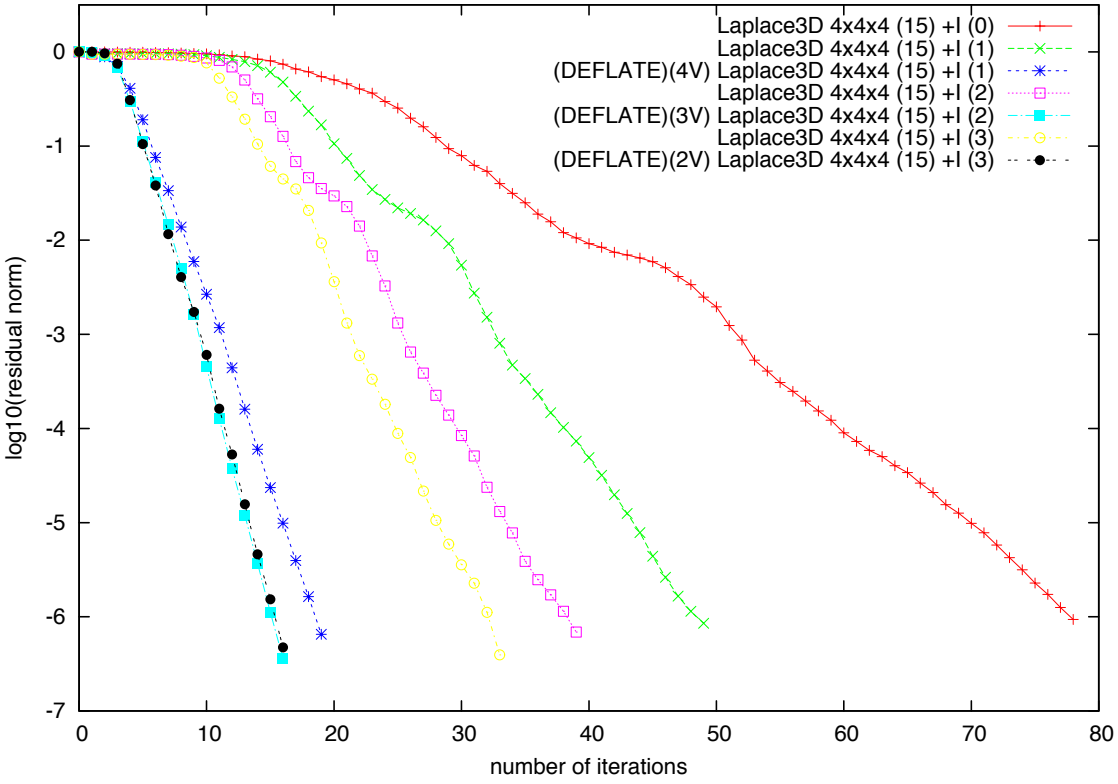
expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	LU[s]	sol[s]
+I(0)	65	584.65		$8.71e-07$			0.25	1.78
+I(1)	41	180.85		$5.34e-09$		0.23	0.26	1.52
D..+I(1)	16	3.36	3	$6.65e-09$	0.04			0.88
+I(2)	31	98.85		$1.21e-08$		0.23+0.35	0.59	1.41
D..+I(2)	15	2.86	2	$3.52e-09$	0.05			1.11
+I(3)	26	64.51		$9.33e-10$		0.2+0.35+0.58	1.31	1.34
D..+I(3)	15	2.48	1	$1.01e-08$	0.04			1.25

Table 5.1: Results for numerical experiment 5.1 where $nx = 4$, $ny = 4$, $nz = 4$ and $m = 10$. For notation see page 118.

Matrix L3D4x4x4n15.mtx

nparts	nrows	nnz
64	226,981	3,316,141



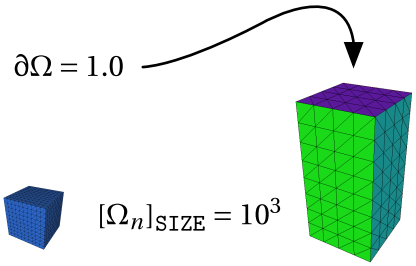


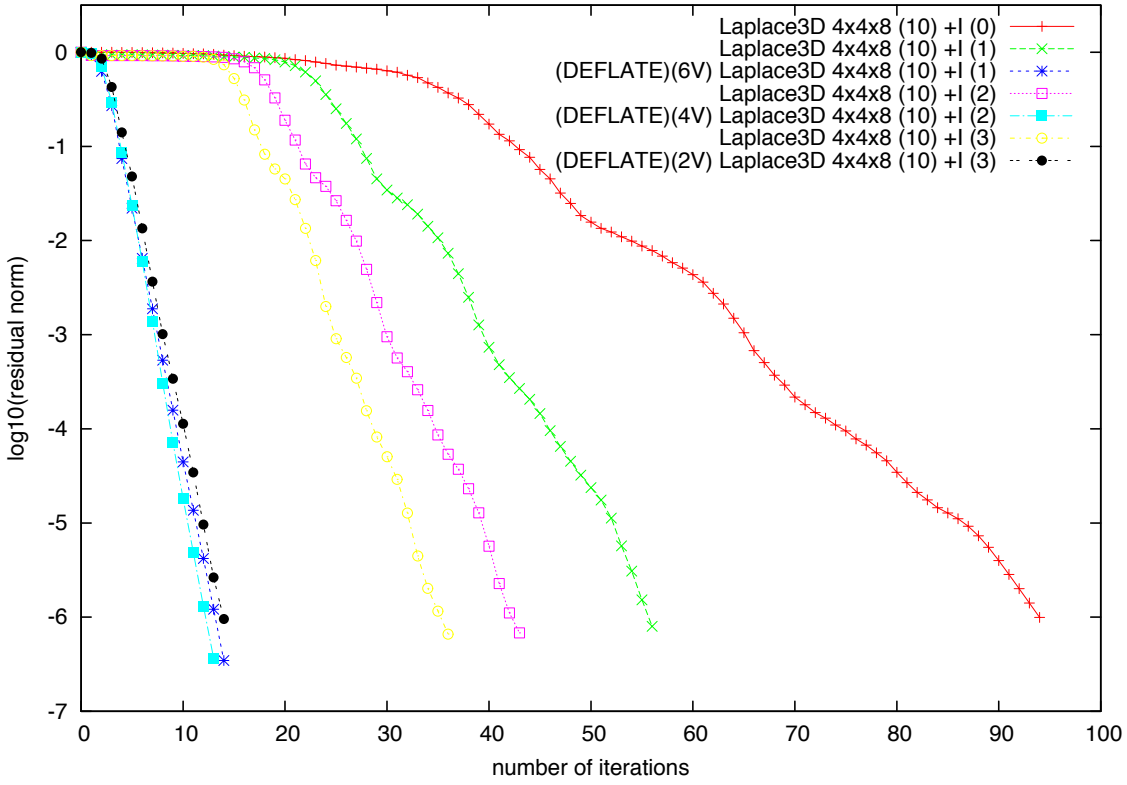
expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	LU[s]	sol[s]
+I(0)	78	879.91		$9.37e-07$			1.46	3.65
+I(1)	49	280.10		$5.41e-09$		1.12	2.49	2.86
D..+I(1)	19	5.49	4	$1.28e-09$	0.27			1.75
+I(2)	39	158.20		$8.67e-10$		1.10+1.60	4.12	2.16
D..+I(2)	16	3.49	3	$9.23e-11$	0.26			2.05
+I(3)	33	106.24		$2.81e-09$		1.07+1.60+2.41	7.14	3.40
D..+I(3)	16	2.98	2	$2.04e-10$	0.24			2.69

Table 5.2: Results for numerical experiment 5.1 where $nx = 4$, $ny = 4$, $nz = 4$ and $m = 15$. For notation see page 118.

Matrix L3D4x4x8n10.mtx

nparts	nrows	nnz
128	136,161	1,976,481



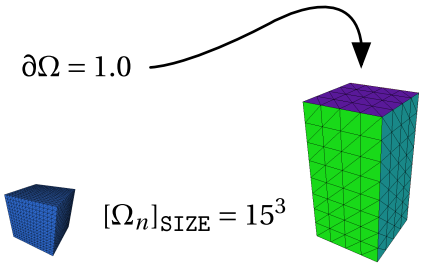


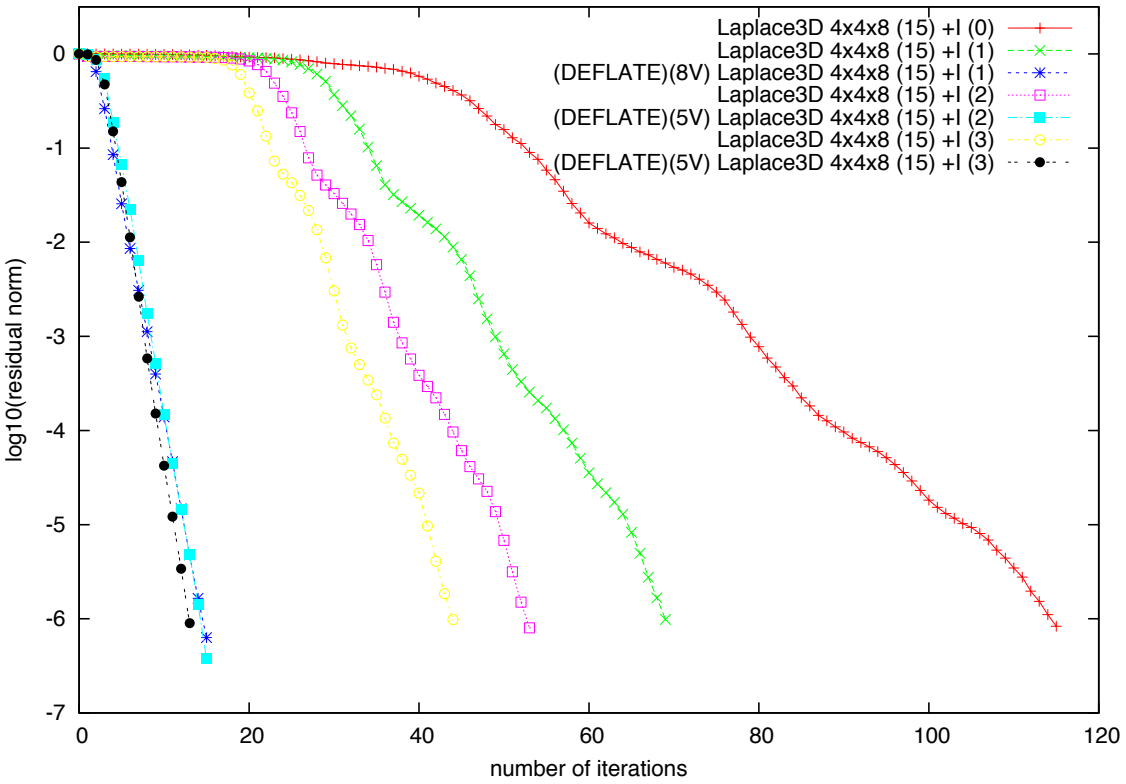
expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS$[s]$	inf$[s]$	LU$[s]$	sol$[s]$
+I(0)	94	2458.56		$9.92e-07$			0.20	9.43
+I(1)	56	759.10		$8.68e-10$		0.26	0.25	4.23
D..+I(1)	14	2.71	6	$8.39e-10$	0.23			2.22
+I(2)	43	414.01		$3.63e-09$		0.25+0.40	0.67	3.86
D..+I(2)	13	2.28	4	$9.31e-10$	0.20			2.01
+I(3)	36	269.63		$4.20e-09$		0.27+0.41+0.73	1.61	3.80
D..+I(3)	14	2.53	2	$7.49e-09$	0.15			2.33

Table 5.3: Results for numerical experiment 5.1 where $nx = 4$, $ny = 4$, $nz = 8$ and $m = 10$. For notation see page 118.

Matrix L3D4x4x8n15.mtx

nparts	nrows	nnz
128	450,241	6,606,721



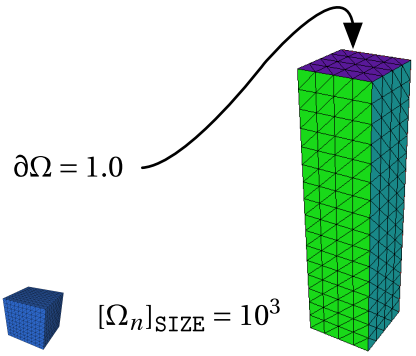


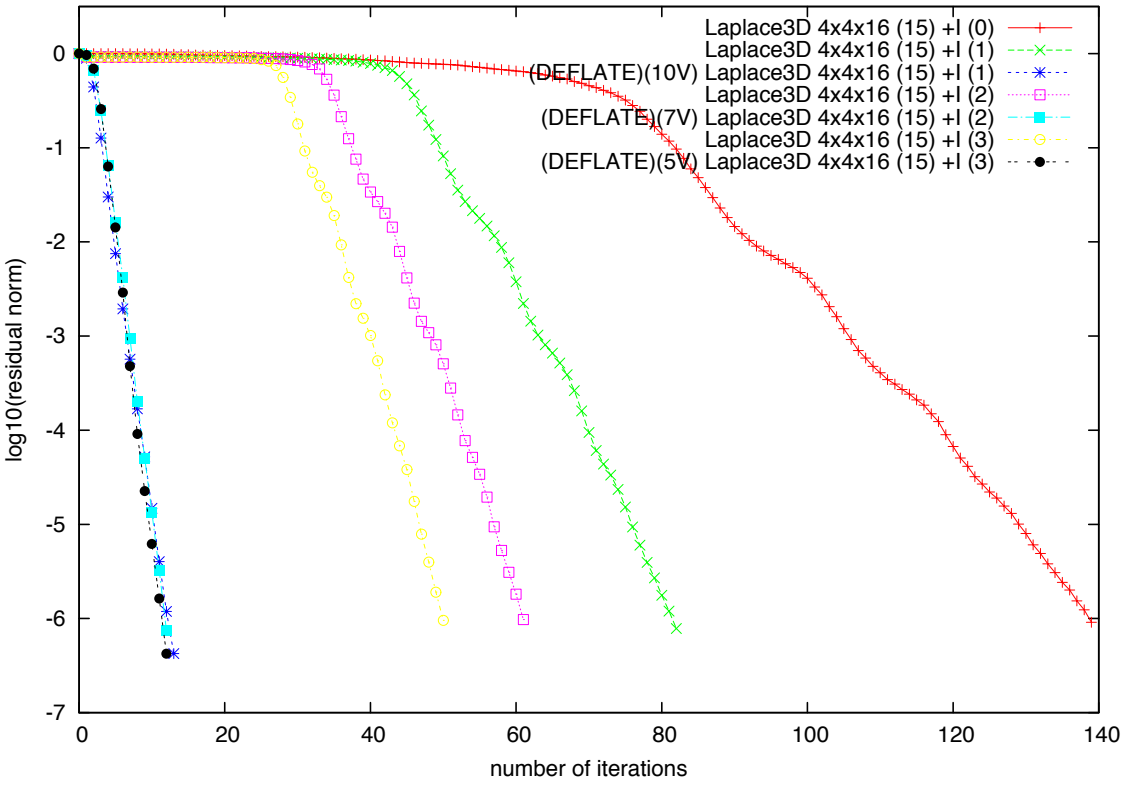
expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	LU[s]	sol[s]
+I(0)	115	3694.10		$8.31e-07$			2.06	8.84
+I(1)	69	1174.49		$1.92e-10$		1.18	4.39	8.35
D..+I(1)	15	3.61	10	$2.55e-09$	1.18			3.36
+I(2)	53	662.34		$3.09e-09$		1.16+1.73	6.35	6.37
D..+I(2)	15	2.78	5	$1.07e-09$	0.76			4.41
+I(3)	44	444.15		$2.59e-09$		1.19+1.76+2.60	14.59	8.03
D..+I(3)	13	2.37	5	$5.46e-09$	1.66			4.91

Table 5.4: Results for numerical experiment 5.1 where $nx = 4$, $ny = 4$, $nz = 8$ and $m = 15$. For notation see page 118.

Matrix L3D4x4x16n10.mtx

nparts	nrows	nnz
256	270,641	3,941,521



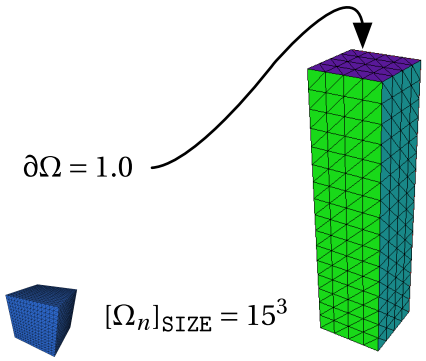


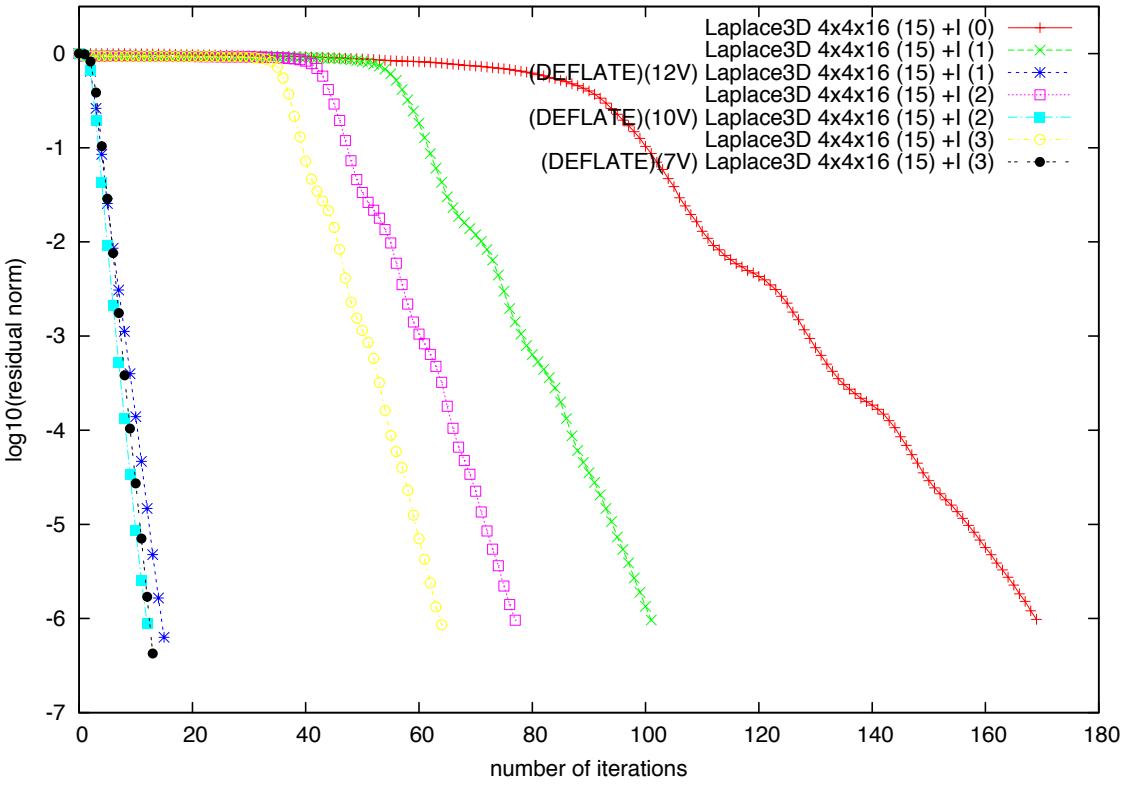
expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	LU[s]	sol[s]
+I(0)	139	10097.30		$9.11e-07$			0.19	12.52
+I(1)	82	3116.28		$5.74e-09$		0.33	0.31	9.27
D..+I(1)	13	2.28	10	$3.69e-09$	0.83			3.74
+I(2)	61	1698.61		$1.50e-09$		0.31+0.46	0.59	7.85
D..+I(2)	12	2.13	7	$5.28e-09$	0.56			3.75
+I(3)	50	1105.62		$3.68e-10$		0.31+0.47+0.72	1.69	6.56
D..+I(3)	12	2.07	5	$2.76e-09$	0.46			3.76

Table 5.5: Results for numerical experiment 5.1 where $nx = 4$, $ny = 4$, $nz = 16$ and $m = 10$. For notation see page 118.

Matrix L3D4x4x16n15.mtx

nparts	nrows	nnz
256	896,761	13,187,881



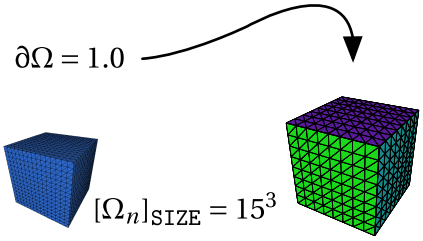


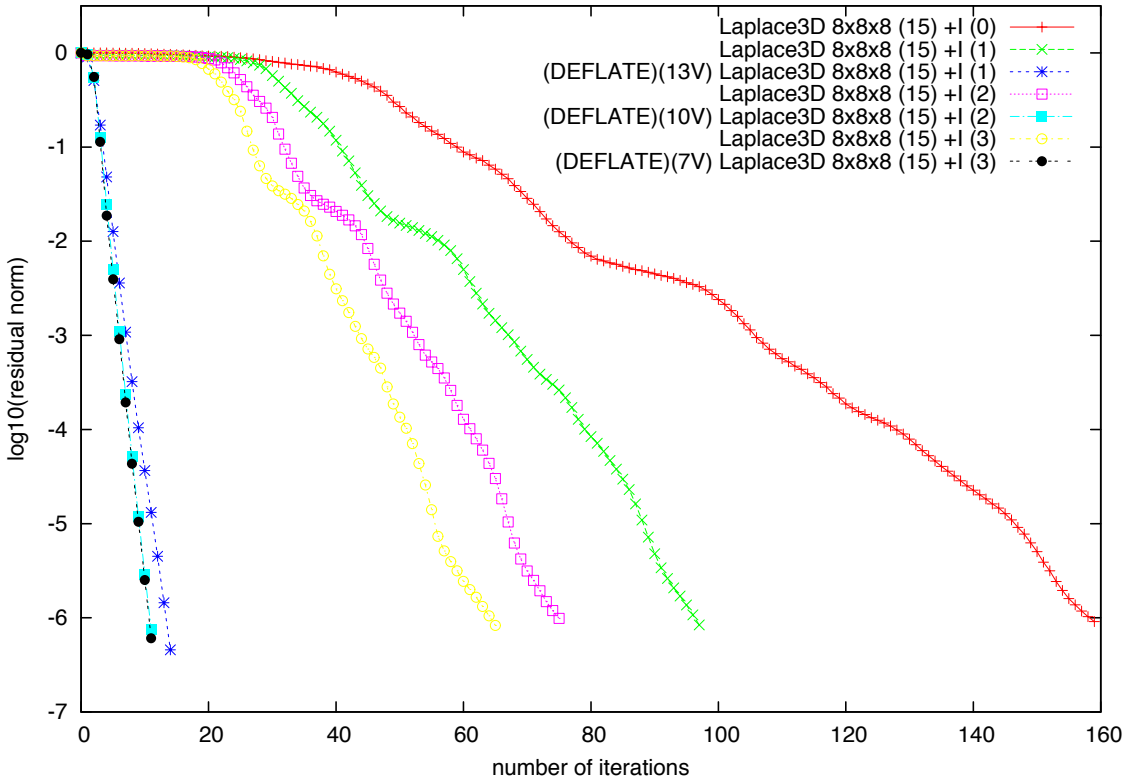
expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	LU[s]	sol[s]
+I(0)	169	15158.70		$9.78e-07$			2.04	17.44
+I(1)	101	4818.07		$1.68e-09$		1.29	3.56	10.30
D..+I(1)	14	5.82	13	$2.18e-06$	2.12			5.17
+I(2)	77	2715.89		$2.12e-09$		1.29+1.89	7.40	12.63
D..+I(2)	12	2.30	10	$9.43e-07$	2.20			5.81
+I(3)	64	1820.48		$2.64e-09$		1.27+1.87+2.74	13.81	14.57
D..+I(3)	13	2.18	7	$1.09e-08$	2.00			6.79

Table 5.6: Results for numerical experiment 5.1 where $nx = 4$, $ny = 4$, $nz = 16$ and $m = 15$. For notation see page 118.

Matrix L3D8x8x8n15.mtx

nparts	nrows	nnz
512	1,771,561	26,223,481



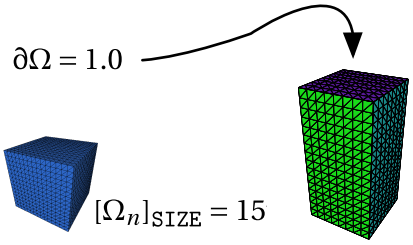


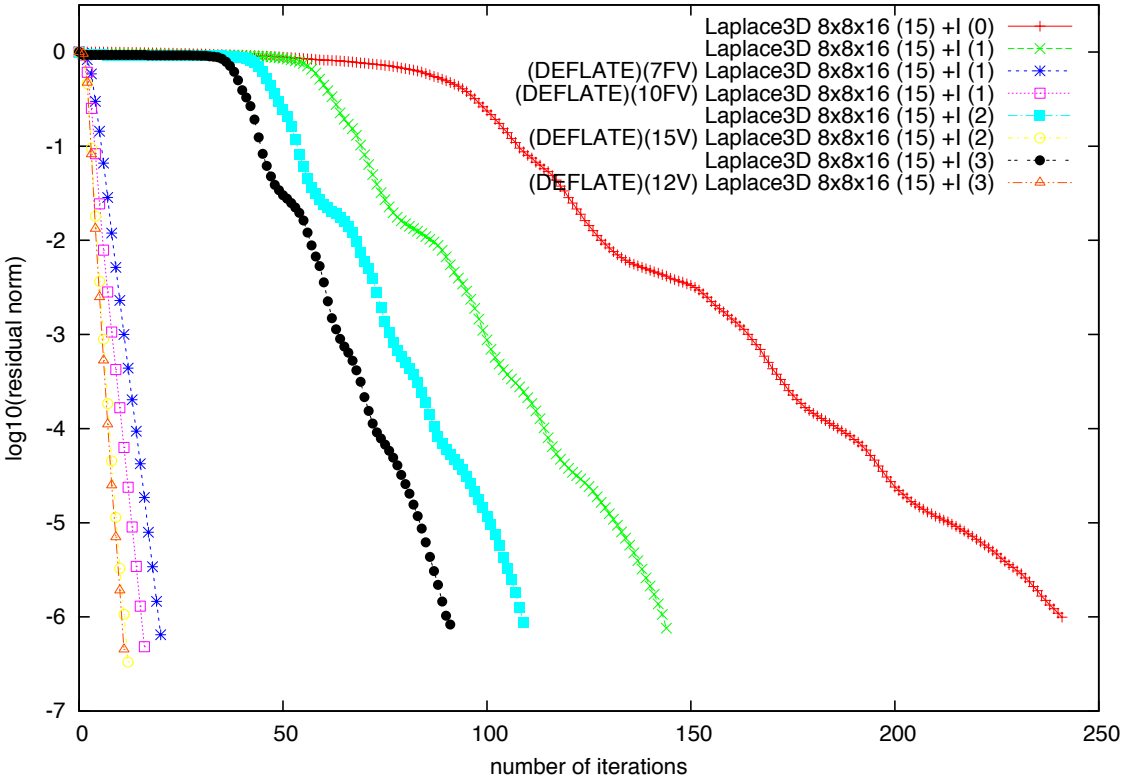
expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	LU[s]	sol[s]
+I(0)	159	4083.11		$9.09e-07$			2.61	16.91
+I(1)	97	1294.14		$1.61e-09$		2.67	4.11	13.91
D..+I(1)	14	3.47	13	$1.05e-07$	3.42			27.56
+I(2)	75	727.35		$3.44e-09$		6.49	8.29	14.10
D..+I(2)	11	2.05	10	$2.37e-09$	2.78			16.56
+I(3)	65	486.40		$1.35e-09$		12.04	14.54	16.13
D..+I(3)	11	1.89	8	$1.33e-09$	2.71			13.83

Table 5.7: Results for numerical experiment 5.1 where $nx = 8$, $ny = 8$, $nz = 8$ and $m = 15$. For notation see page 118.

Matrix L3D8x8x16n15.mtx

nparts	nrows	nnz
1024	3,528,481	52,345,441





expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	LU[s]	sol[s]
+I(0)	241	16715.00		$9.95e-07$			7.46	56.51
+I(1)	144	5296.63		$9.07e-10$		6.07	16.24	48.06
D..+I(1)	20	5.44	7F	$1.38e-06$	7.35			44.32
D..+I(1)	16	4.26	10F	$1.22e-05$	12.79			73.38
+I(2)	109	2975.54		$1.68e-09$		14.94	16.71	41.26
D..+I(2)	12	2.76	15	$1.22e-05$	16.15			111.32
+I(3)	91	1988.98		$9.24e-10$		27.85	29.01	33.43
D..+I(3)	11	2.23	12	$1.47e-05$	12.61			57.62

Table 5.8: Results for numerical experiment 5.1 where $nx = 8$, $ny = 8$, $nz = 16$ and $m = 15$. For notation see page 118.

5.2 Algebraic Multi Grid method as a sub-solver in ADDM

Numerical Experiment 5.2. *In this investigation we consider the same problem as in experiment 5.1, but we limit our tests only to the variants with the biggest sub-domain size i.e.,* $[\Omega_i]_{SIZE} = 15^3$. *Other changes are due to setup of the experiment.*

5.2.1 Setup of the Experiment 5.2

The only change in setup of numerical experiment 5.2 in comparison to previous tests is a type of sub-solver used in "Schwarz preconditioner" of the first solve. Instead of `SuperLU` routines we used `HYPRE Algebraic Multi Grid` methods via interface of `PETSc` library with the default setup (convergence tolerance $tol = 1e^{-6}$). In a consequence of using `AMG` instead of direct solver we changed also a type of first iterative method from GMRES to F(*lexible*)GMRES. Thus we applied Schwarz Method as a right preconditioner and approximated eigenvectors needed special treatment in order to by successfully used in coarse space computation for the two-level preconditioner (see §3.2 p. 72).

The stages of experiment 5.2 are depicted on figure 5.2.1.

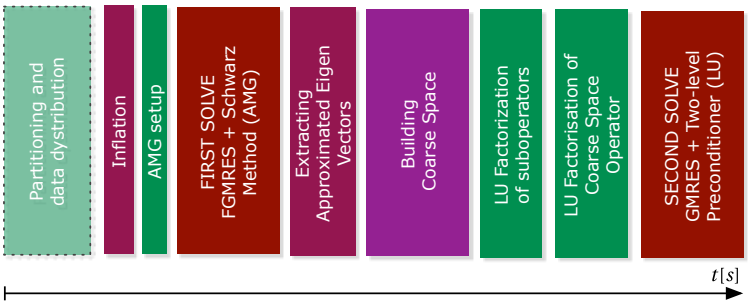


Figure 5.2: Stages of experiment 5.1 on time axis.

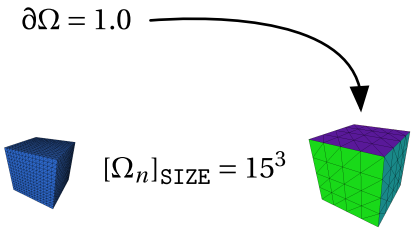
Notation 5.2. *Because we changed sub-solver type, instead of LU factorisation time in first solve we measured time of configuration AMG preconditioners for endomorphic* Partial Operators*, which we denoted in tables by* ***AMG****.*

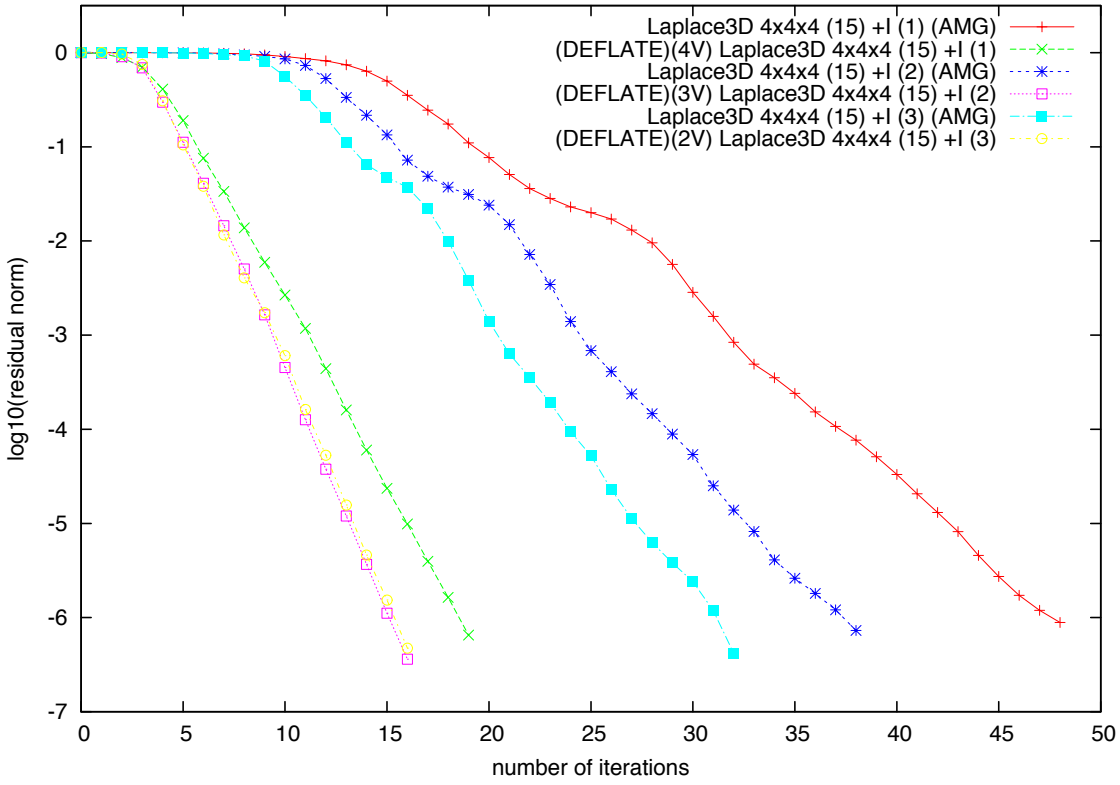
AMG$[s]$	*configuration time of iterative solver with AMG preconditioner for each* `Partial Operator` *in* `DDMOperator`

Time of LU factorisation of `Partial Operators` *in second solve was automatically included to time of coarse space computation (****CS****$[s]$).*

Matrix L3D4x4x4n15.mtx

nparts	nrows	nnz
64	226,981	3,316,141



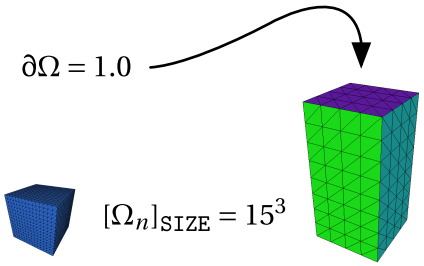


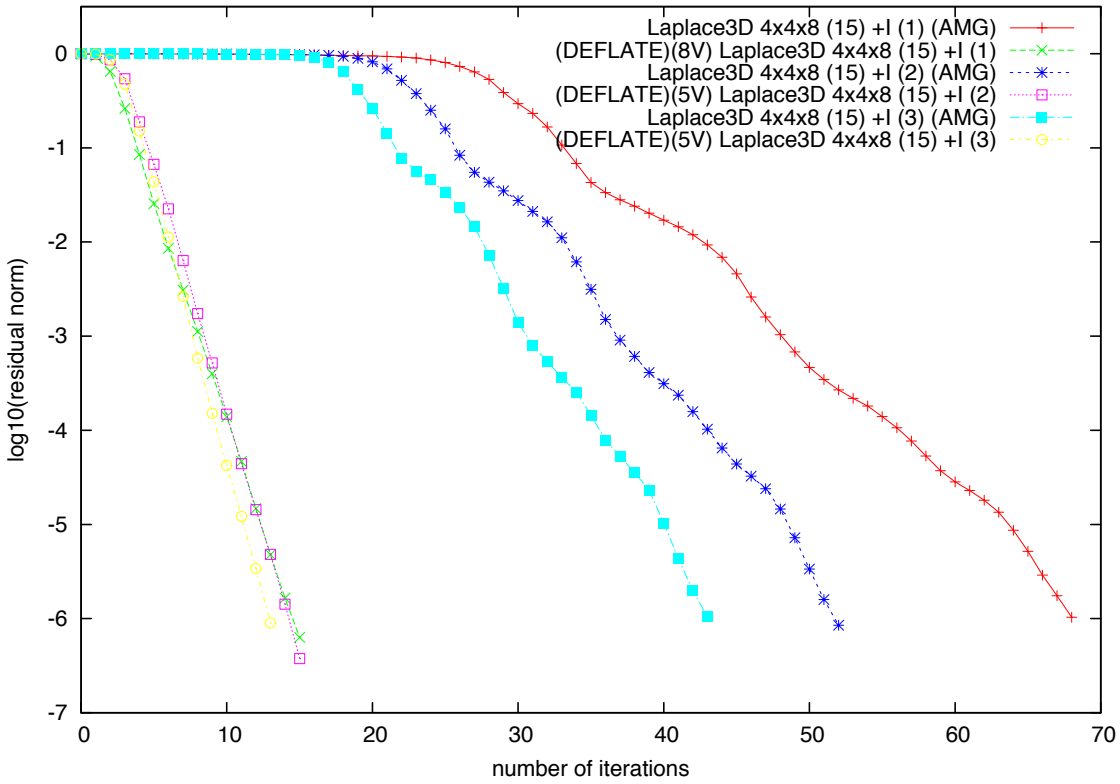
expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	AMG[s]	sol[s]
+I(1)	43	280.11		$5.41e-09$		1.07	0.24	3.77
D..+I(1)	19	5.81	4	$1.90e-09$	0.31			1.85
+I(2)	33	158.20		$8.49e-10$		1.05+1.64	0.18	4.38
D..+I(2)	16	3.40	3	$9.20e-11$	0.28			2.14
+I(3)	28	106.24		$2.74e-09$		1.04+1.62+2.45	0.22	5.68
D..+I(3)	16	2.91	2	$2.17e-10$	0.25			2.66

Table 5.9: Results for numerical experiment 5.2 where $nx = 4$, $ny = 4$, $nz = 4$ and $m = 15$. For notation see page 118 and 128.

Matrix L3D4x4x8n15.mtx

nparts	nrows	nnz
128	450,241	6,606,721



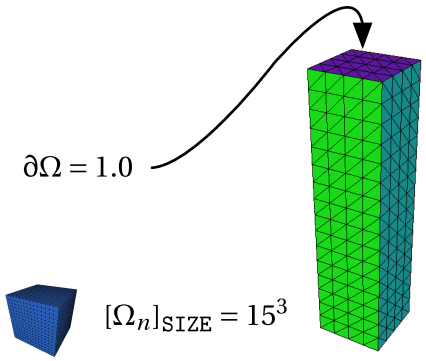


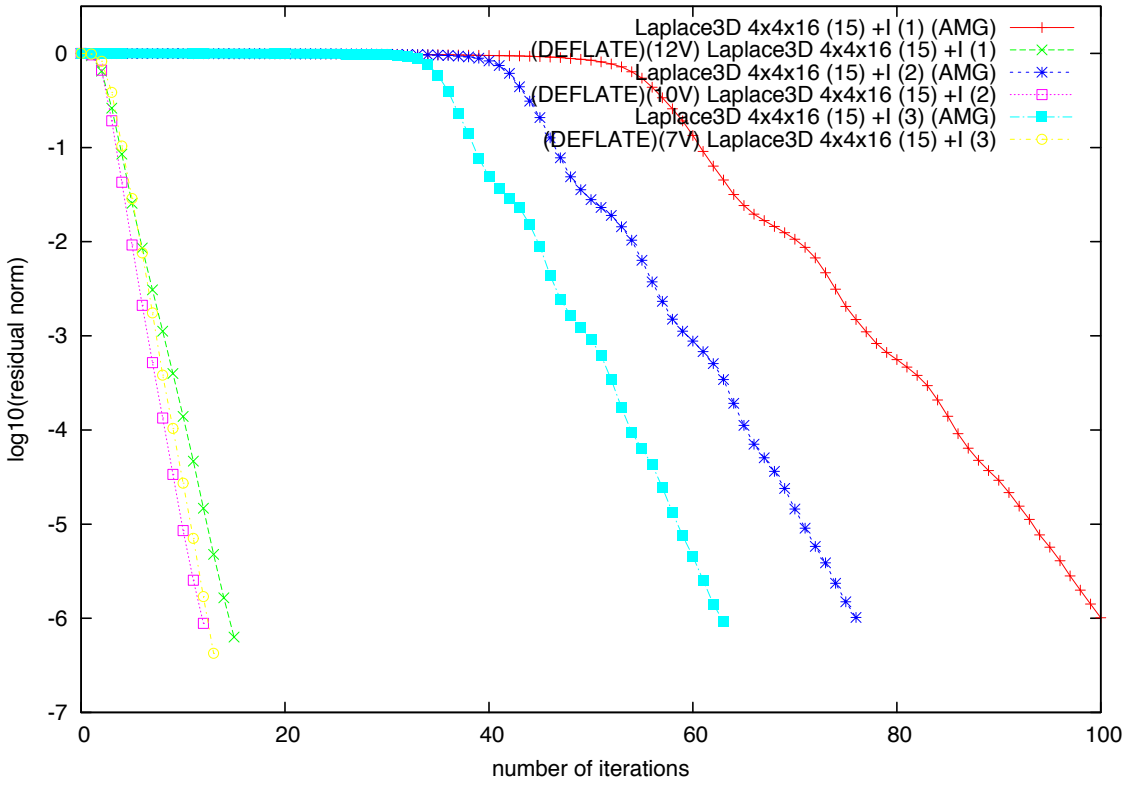
expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	AMG[s]	sol[s]
+I(1)	69	1174.49		$1.13e-10$		1.14	0.16	4.83
D..+I(1)	15	2.6	9	$9.13e-09$	1.20			3.37
+I(2)	53	662.34		$3.03e-09$		1.10+2.54	0.18	7.02
D..+I(2)	15	2.73	5	$6.37e-09$	0.84			4.37
+I(3)	44	444.16		$2.61e-09$		1.11+1.69+2.52	0.25	8.20
D..+I(3)	13	2.32	5	$8.19e-09$	1.75			4.84

Table 5.10: Results for numerical experiment 5.2 where $nx = 4$, $ny = 4$, $nz = 8$ and $m = 15$. For notation see page 118 and 128.

Matrix L3D4x4x16n15.mtx

nparts	nrows	nnz
256	896,761	13,187,881





expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	AMG[s]	sol[s]
+I(1)	101	4818.07		$1.85e-09$		1.22	0.20	4.83
D..+I(1)	13	4.53	12	$1.63e-06$	2.24			5.17
+I(2)	77	2715.89		$1.62e-09$		1.25+1.86	0.20	7.02
D..+I(2)	10	2.31	10	$1.78e-08$	2.30			5.88
+I(3)	64	1820.48		$2.24e-09$		1.25+1.88+2.81	0.23	8.20
D..+I(3)	7	2.18	7	$1.52e-08$	2.11			6.65

Table 5.11: Results for numerical experiment 5.2 where $nx = 4$, $ny = 4$, $nz = 16$ and $m = 15$. For notation see page 118 and 128.

5.3 Real test cases

The following section is dedicated to real test cases which originated from simulations of various petroleum and geophysical problems. Problems are organised in set of sparse matrices (of a different size and sparsity pattern) and corresponding vector with righthand side.

As in the experiment 5.1 we solve a given linear system twice, i.e., once with a one-level preconditioner and after restart with two-level preconditioner. The only difference is that instead of manual partition we used sequential graph partitioner (METIS 5.1 [36]). We refer to §5.1.1 for most of the setup details.

5.3.1 IFP Matrix Collection - pressure block only

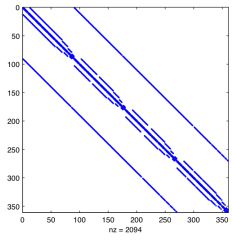
In this subsection we consider the scalar problems (pressure unknown only). For partitioning we used *kway* subroutine from METIS library, but we repeated experiments also for matrix partitioned with weights defined on edges of its adjacency graph (see §2.4 p. 50). In description we used the same notation as for previous experiments (see §5.1 p. 118).

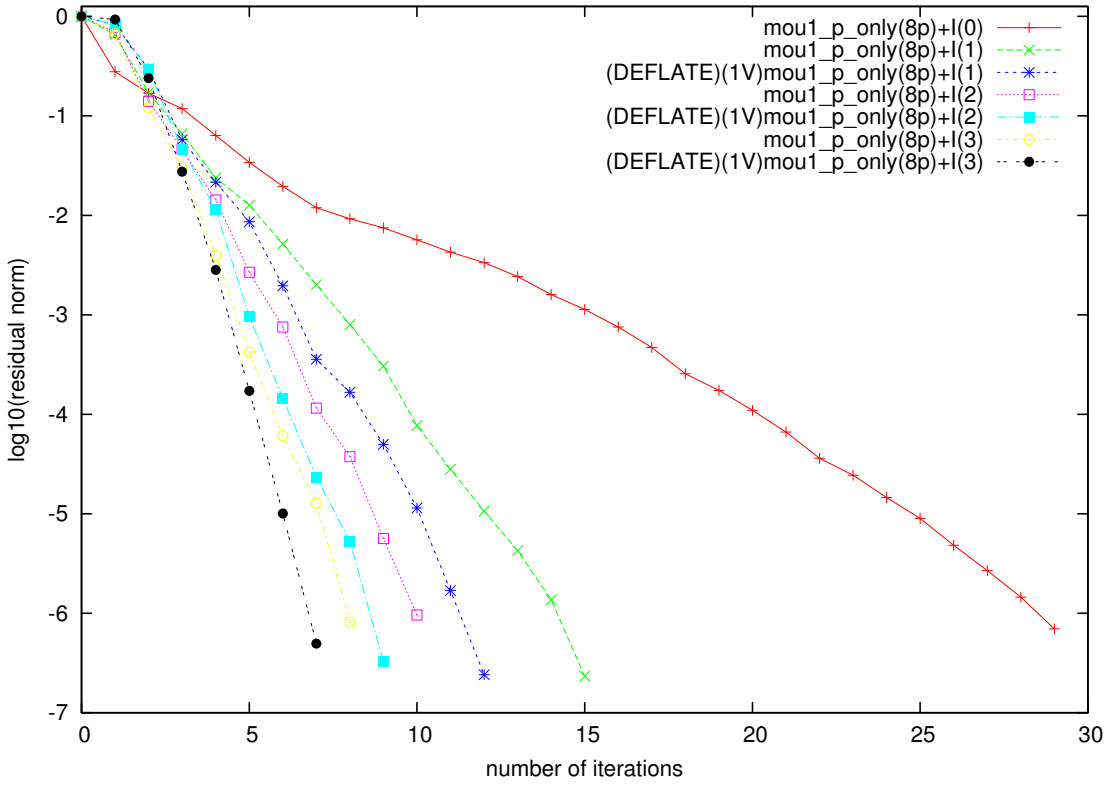
name	**n-parts**	**nrows**	**nnz**
`mou1`	8	360	2.094
`Canta`	16	8.016	60.246
`IvaskBO`	32	49.572	478.050
`IvaskMULTI`	32	49.572	480.612
`GCS`	128	370.982	2.916.372
`spe10`	256	1.094.421	7.515.591

Table 5.12: Matrixes used in experiment, where: **nrow** - number of rows, **n-parts** - number of parts (subdomains), **nnz** - number of non-zero elements.

Matrix `mou1_p_only.mtx`

nparts	nrows	nnz
8	360	2,094



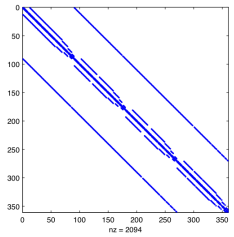


expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	LU[s]	sol[s]
+I(0)	29	24.84		$6.95e-07$			0.00	0.01
+I(1)	15	5.13		$6.99e-09$		0.00	0.00	0.02
D..+I(1)	12	3.48	1F	$1.27e-09$	0.00			0.00
+I(2)	10	2.53		$6.96e-08$		0.00 + 0.00	0.00	0.02
D..+I(2)	9	1.89	1F	$8.30e-09$	0.00			0.01
+I(3)	8	1.51		$1.30e-10$		0.01 + 0.00 +0.00	0.00	0.01
D..+I(3)	7	1.32	1F	$3.21e-10$	0.01			0.00

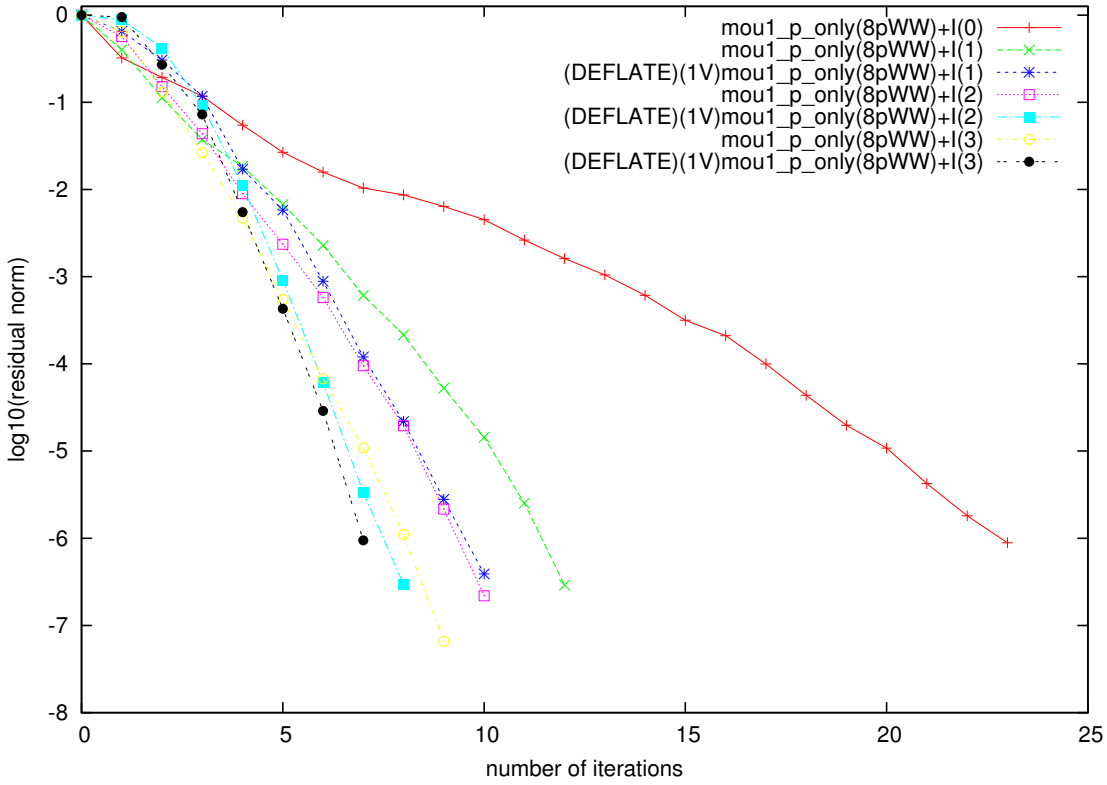
Table 5.13: Results for the matrix: `mou1_p_only.mtx`.

Matrix mou1_p_only.mtx

nparts	nrows	nnz
8	360	2,094



(METIS partitioner with weights)

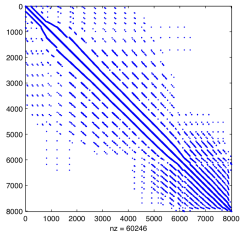


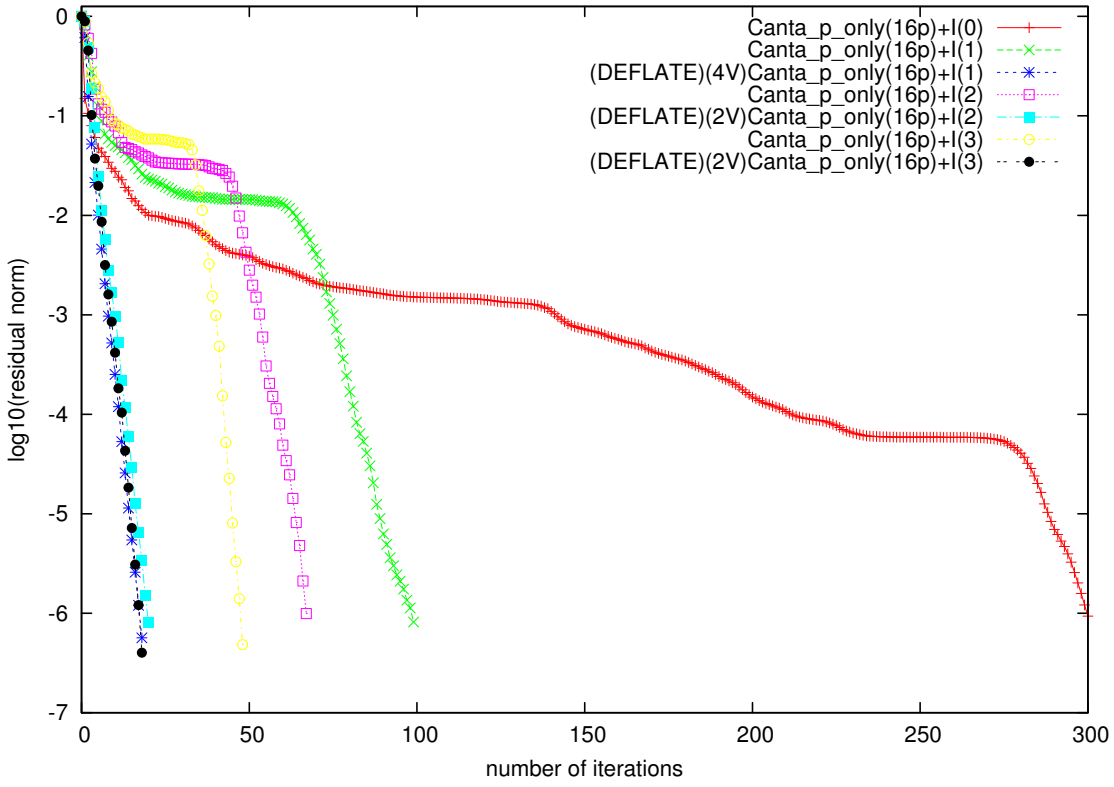
expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	LU[s]	sol[s]
+I(0)	23	18.04		$8.86e-07$			0.00	0.03
+I(1)	12	3.75		$1.63e-08$		0.00	0.00	0.01
D..+I(1)	10	2.49	1F	$1.99e-08$	0.00			0.01
+I(2)	10	2.32		$2.62e-09$		0.00+0.01	0.00	0.02
D..+I(2)	8	1.66	1F	$4.52e-09$	0.00			0.00
+I(3)	9	1.60		$1.72e-09$		0.00+0.01+0.00	0.00	0.02
D..+I(3)	7	1.26	1F	$2.80e-08$	0.00			0.00

Table 5.14: Results for the matrix: mou1_p_only.mtx (partitioner with weight).

Matrix Canta_p_only.mtx

nparts	nrows	nnz
16	8,016	60,246



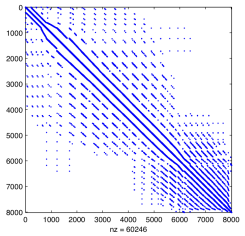


expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	LU[s]	sol[s]
+I(0)	300	351011		$9.34e-07$			0.01	3.08
+I(1)	99	18149.10		$3.78e-10$		0.01	0.01	0.76
D..+I(1)	18	6.90	12	$4.06e-06$	0.02			0.06
+I(2)	67	7436.2		$7.47e-10$		0.01+0.04	0.01	0.12
D..+I(2)	20	7.98	6	$1.20e-09$	0.01			0.02
+I(3)	48	3872.88		$9.33e-10$		0.02+0.03+0.05	0.03	0.11
D..+I(3)	18	7.47	4	$6.61e-10$	0.01			0.03

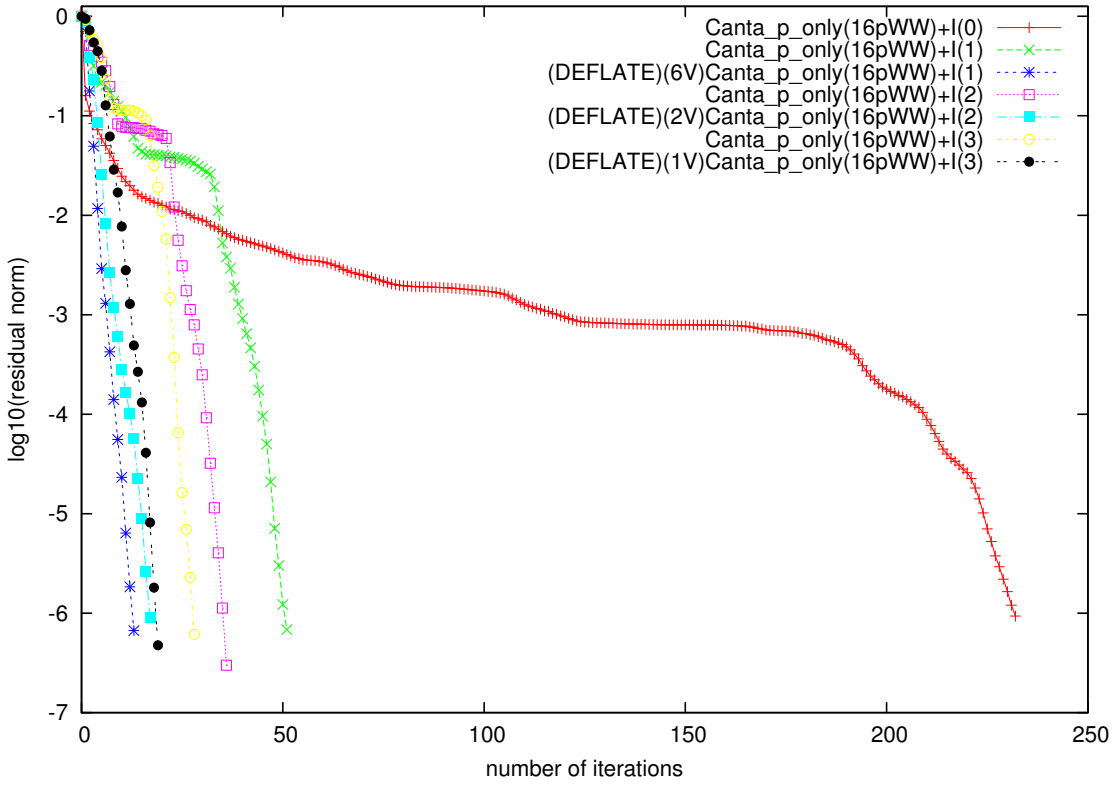
Table 5.15: Results for the matrix: Canta_p_only.mtx.

Matrix Canta_p_only.mtx

nparts	nrows	nnz
16	8,016	60,246



(METIS partitioner with weights)

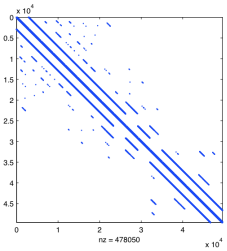


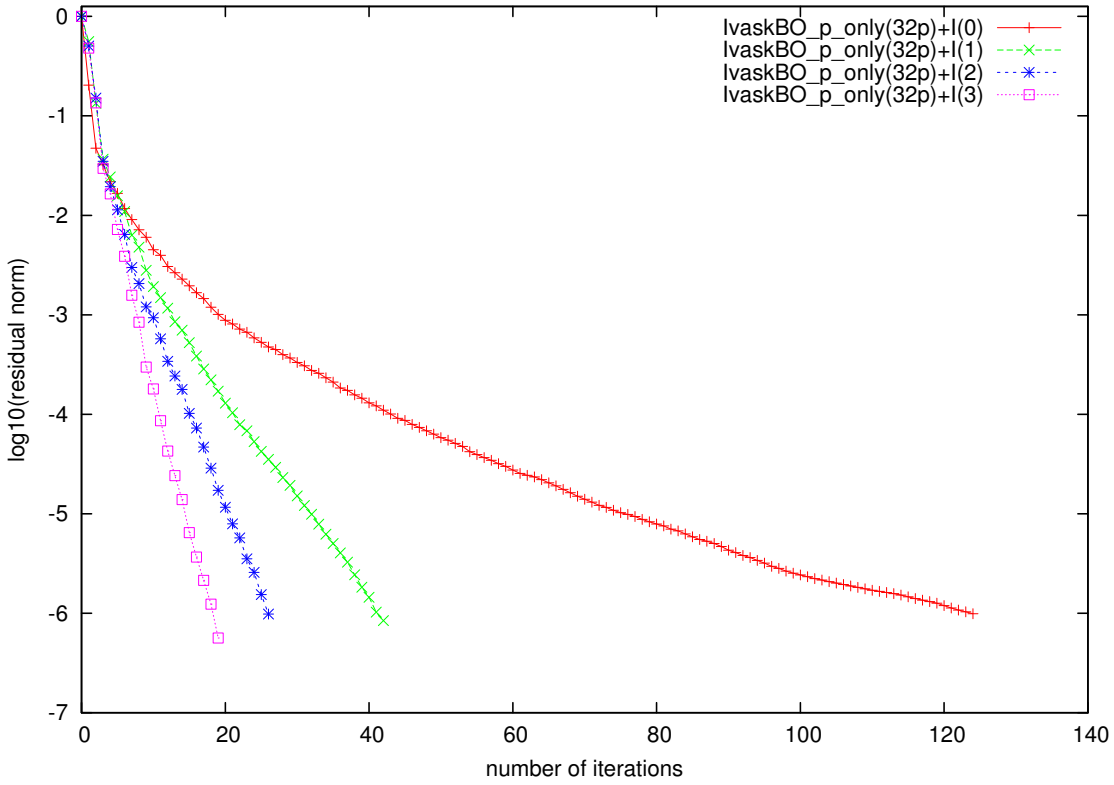
expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	LU[s]	sol[s]
+I(0)	232	68506.6		$9.35e-07$			0.00	2.45
+I(1)	51	5623.23		$5.85e-10$		0.04	0.01	0.13
D..+I(1)	13	3.64	6	$2.36e-10$	0.00			0.03
+I(2)	36	2702.11		$6.81e-09$		0.05+0.07	0.02	0.11
D..+I(2)	17	7.88	2	$1.05e-09$	0.00			0.03
+I(3)	28	1492.98		$6.65e-11$		0.05+0.06+0.10	0.04	0.13
D..+I(3)	19	12.39	1	$2.34e-09$	0.01			0.04

Table 5.16: Results for the matrix: Canta_p_only.mtx (partitioner with weight).

Matrix IvaskBO_p_only.mtx

nparts	nrows	nnz
32	49,572	478,050



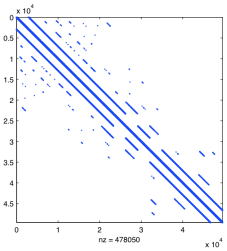


expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	LU[s]	sol[s]
+I(0)	124	1348.60		$9.89e-07$			0.12	0.84
+I(1)	42	65.53		$1.85e-09$		0.10	0.2	0.28
D..+I(1)	-	-					*err!*	
+I(2)	26	20.60		$2.92e-09$		0.10+0.14	0.37	0.34
D..+I(2)	-	-					*err!*	
+I(3)	19	10.45		$2.34e-09$		0.11+0.14+0.21	0.49	0.54
D..+I(3)	-	-					*err!*	

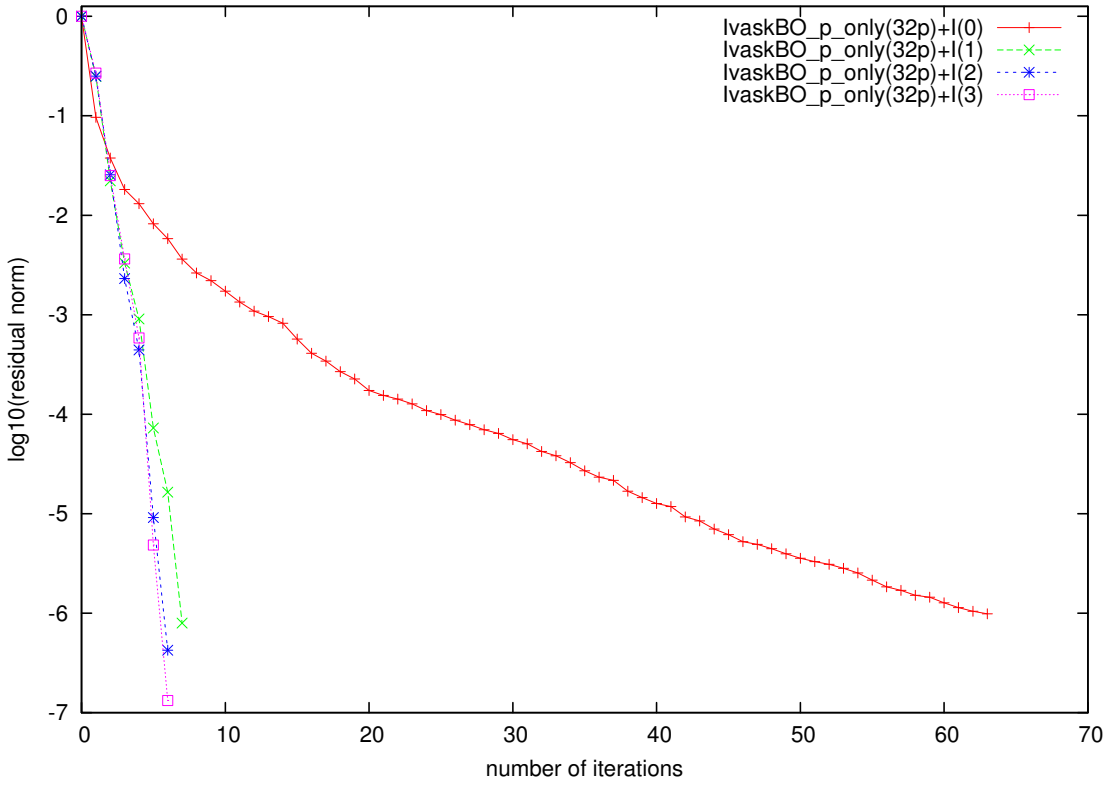
Table 5.17: Results for the matrix: IvaskBO_p_only.mtx.

Matrix IvaskBO_p_only.mtx

nparts	nrows	nnz
32	49,572	478,050



(METIS partitioner with weights)

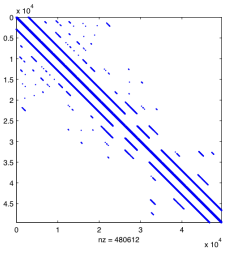


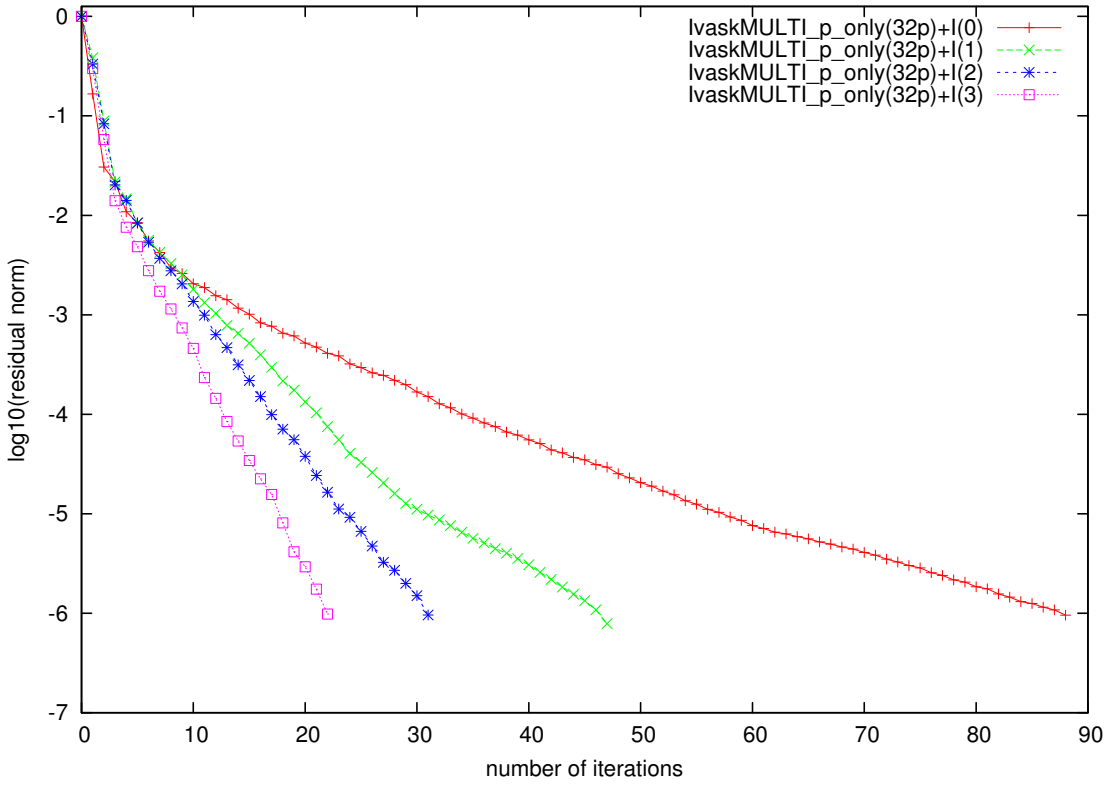
expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	LU[s]	sol[s]
+I(0)	63	249.86		$9.85e-07$			0.06	0.24
+I(1)	7	2.49		$9.70e-10$		0.30	0.14	0.25
D..+I(1)	-						*err!*	
+I(2)	6	2.31		$4.09e-09$		0.29+0.42	0.27	0.47
D..+I(2)	-						*err!*	
+I(3)	6	2.15		$1.41e-09$		0.30+0.42+0.55	0.4	1.05
D..+I(3)	-						*err!*	

Table 5.18: Results for the matrix: IvaskBO_p_only.mtx (partitioner with weight).

Matrix IvaskMULTI_p_only.mtx

nparts	nrows	nnz
32	49,572	480,612



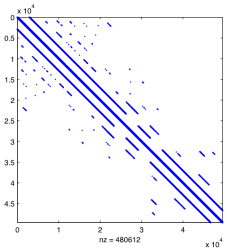


expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	LU[s]	sol[s]
+I(0)	88	516.69		$9.58e-07$			0.14	0.47
+I(1)	47	131.22		$7.24e-09$		0.13	0.24	0.27
D..+I(1)	-	-					*err!*	
+I(2)	31	33.75		$3.67e-09$		0.12+0.16	0.39	0.41
D..+I(2)	-	-					*err!*	
+I(3)	22	17.84		$1.80e-08$		0.11+0.15+0.24	0.56	0.67
D..+I(3)	-	-					*err!*	

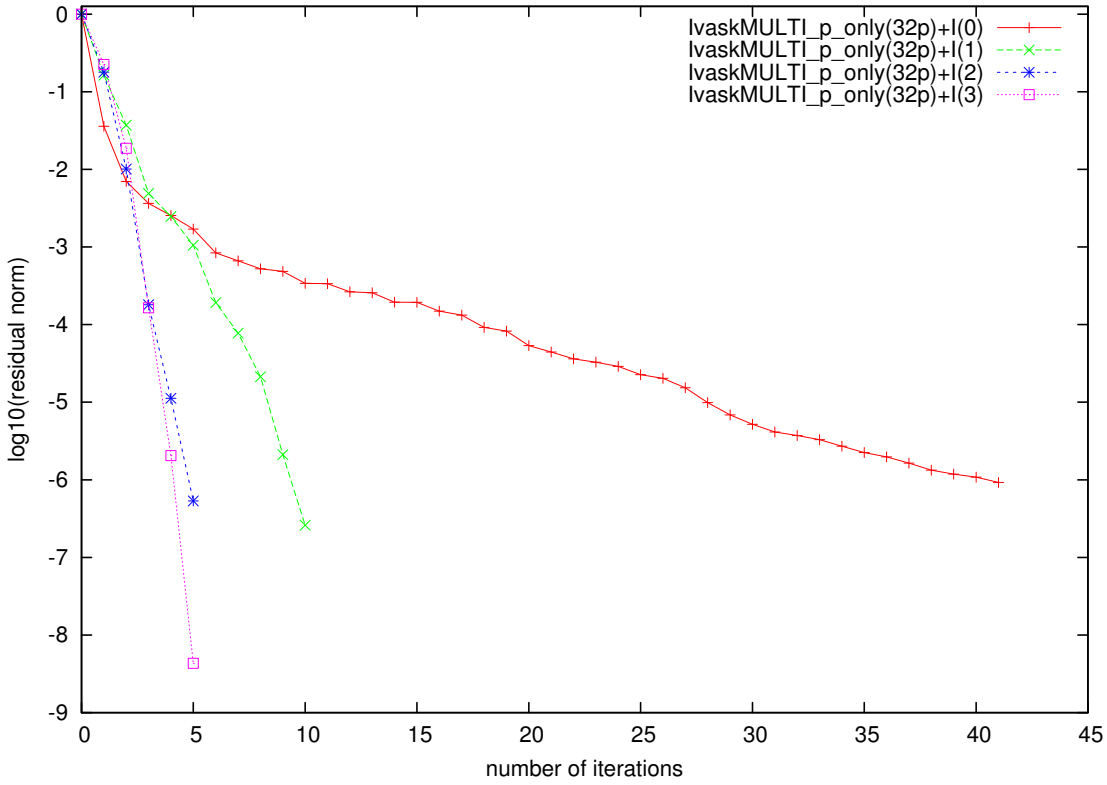
Table 5.19: Results for the matrix: IvaskMULTI_p_only.mtx.

Matrix IvaskMULTI_p_only.mtx

nparts	nrows	nnz
32	49,572	480,612



(METIS partitioner with weights)

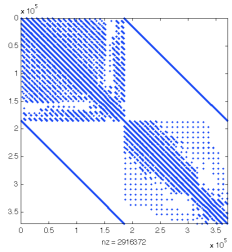


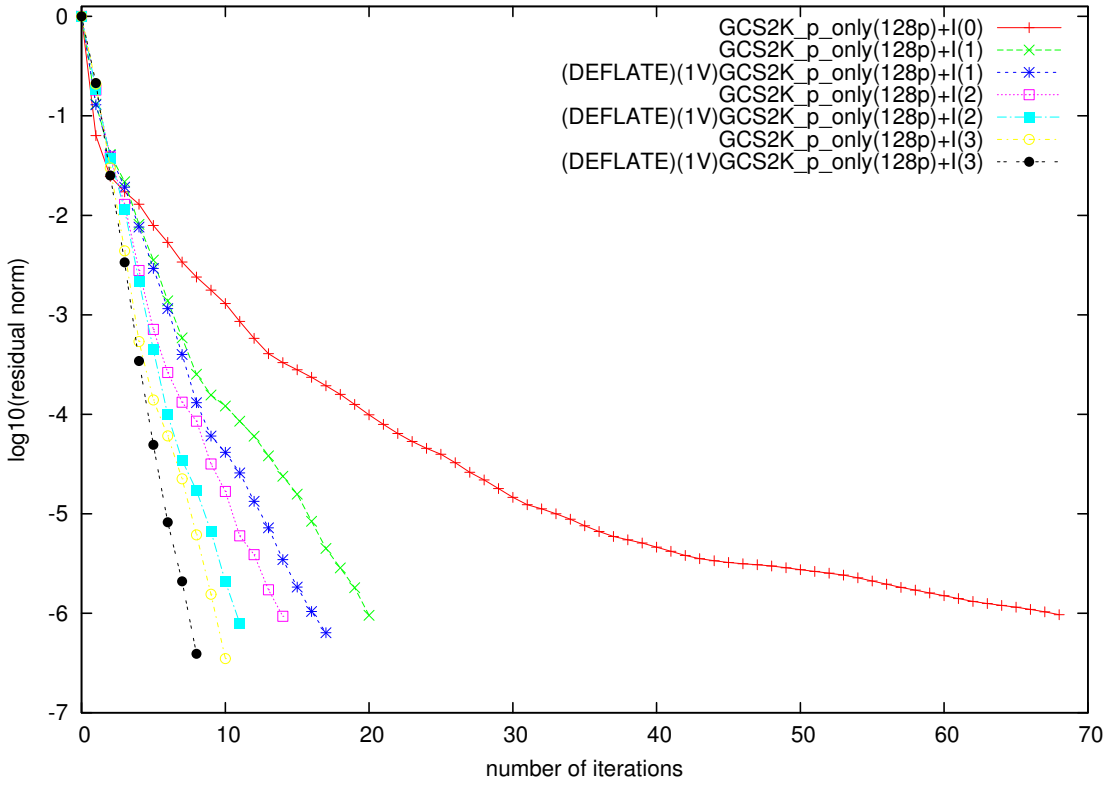
expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	LU[s]	sol[s]
+I(0)	41	117.50		$9.27e-07$			0.09	0.16
+I(1)	10	5.08		$2.34e-09$		0.23	0.19	0.29
D..+I(1)	-						*err!*	
+I(2)	5	1.04		$5.94e-09$		0.23+0.26	0.29	0.55
D..+I(2)	-						*err!*	
+I(3)	5	1.04		$1.58e-11$		0.22+0.26+0.31	0.55	1.12
D..+I(3)	-						*err!*	

Table 5.20: Results for the matrix: IvaskMULTI_p_only.mtx (partitioner with weight).

Matrix GCS_p_only.mtx

nparts	nrows	nnz
128	370,982	2,916,372



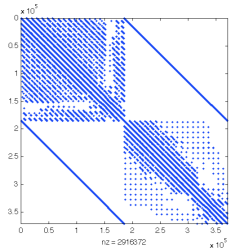


expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	LU[s]	sol[s]
+I(0)	68	803.04		$9.66e-07$			0.64	2.06
+I(1)	20	18.77		$3.09e-09$		1.25	1.58	2.43
D..+I(1)	17	10.24	1F	$1.31e-09$	0.03			0.27
+I(2)	14	7.72		$1.29e-09$		1.27+1.11	7.22	1.22
D..+I(2)	11	4.53	1F	$1.03e-09$	0.07			0.27
+I(3)	10	3.04		$8.96e-10$		1.29+1.11+1.66	12.22	2.03
D..+I(3)	8	1.88	1F	$5.31e-10$	0.10			0.29

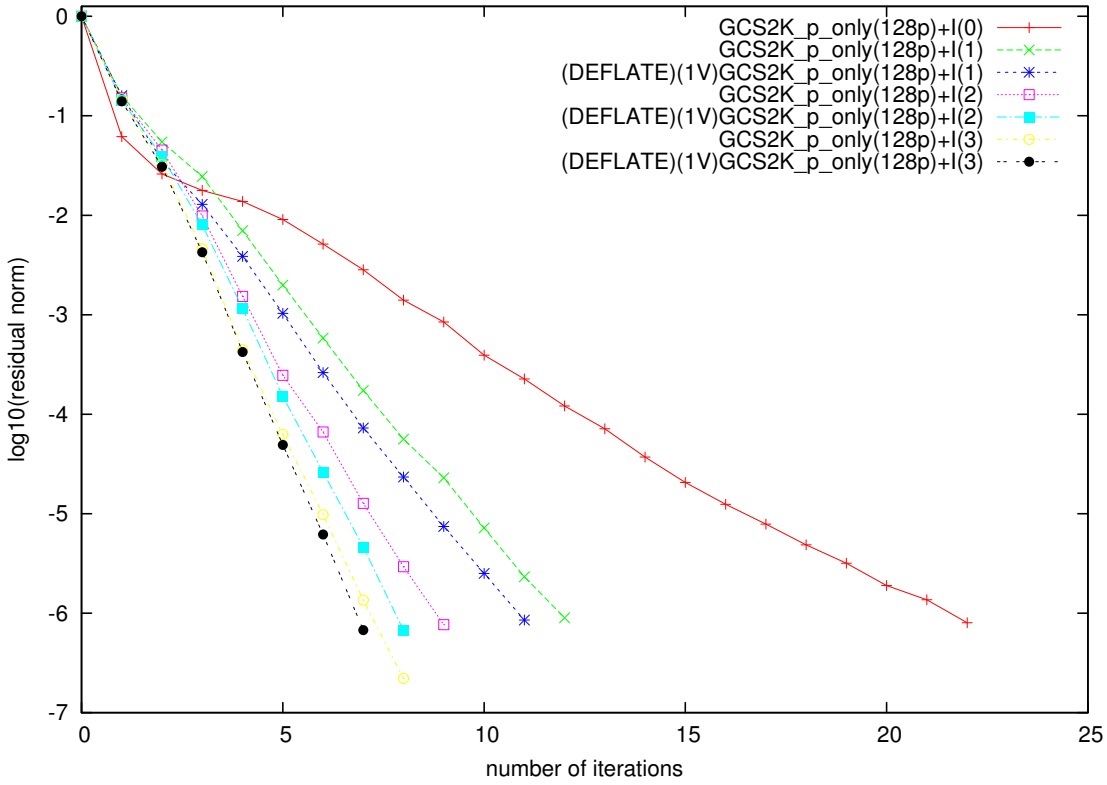
Table 5.21: Results for the matrix: GCS_p_only.mtx.

Matrix GCS_p_only.mtx

nparts	nrows	nnz
128	370,982	2,916,372



(METIS partitioner with weights)

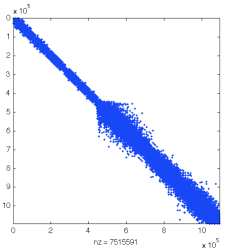


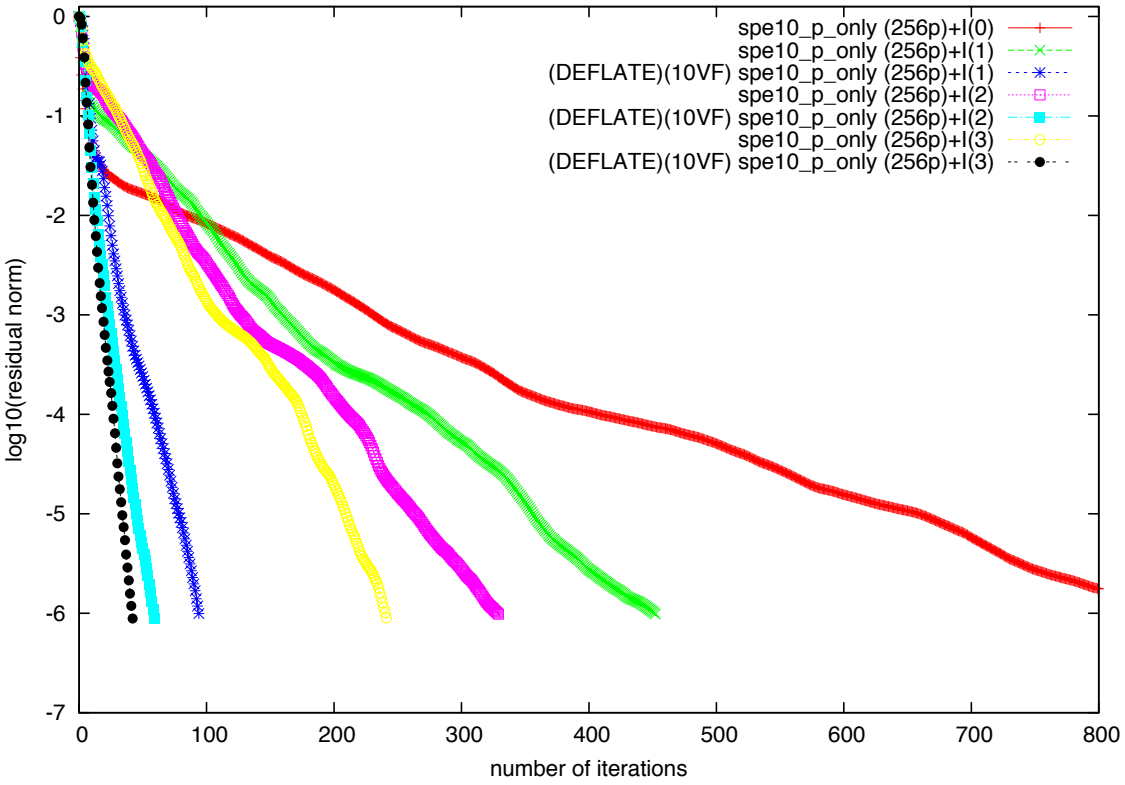
expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	LU[s]	sol[s]
+I(0)	22	14.50		$8.00e-07$			0.59	0.71
+I(1)	12	3.82		$2.36e-09$		2.61	2.26	2.19
D..+I(1)	11	3.14	1F	$5.47e-09$	0.03			0.18
+I(2)	9	2.11		$5.30e-09$		2.59+3.98	5.97	4.14
D..+I(2)	8	1.85	1F	$1.46e-09$	0.06			0.25
+I(3)	8	1.59		$1.49e-09$		2.66+3.99+6.16	13.69	7.75
D..+I(3)	7	1.44	1F	$9.29e-09$	0.12			0.34

Table 5.22: Results for the matrix: GCS_p_only.mtx (partitioner with weight).

Matrix `spe10_p_only.mtx`

nparts	nrows	nnz
256	1,094,421	7,515,591



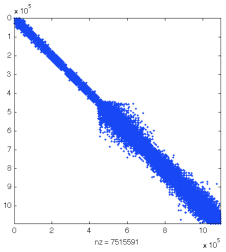


expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	LU[s]	sol[s]
+I(0)	800	87727.30		$3.87e-06$			3.03	117.05
+I(1)	453	15179.60		$1.48e-09$		2.90	5.75	60.66
D..+I(1)	81	120.14	10F	$4.78e-08$	1.15			8.86
+I(2)	295	7821.55		$3.99e-09$		2.89+3.56	10.67	34.16
D..+I(2)	60	98.95	10F	$2.05e-08$	5.72			8.06
+I(3)	232	5056.57		$4.58e-09$		2.91+5.82+4.96	19.32	32.08
D..+I(3)	40	23.82	10F	$2.21e-08$	4.01			7.2

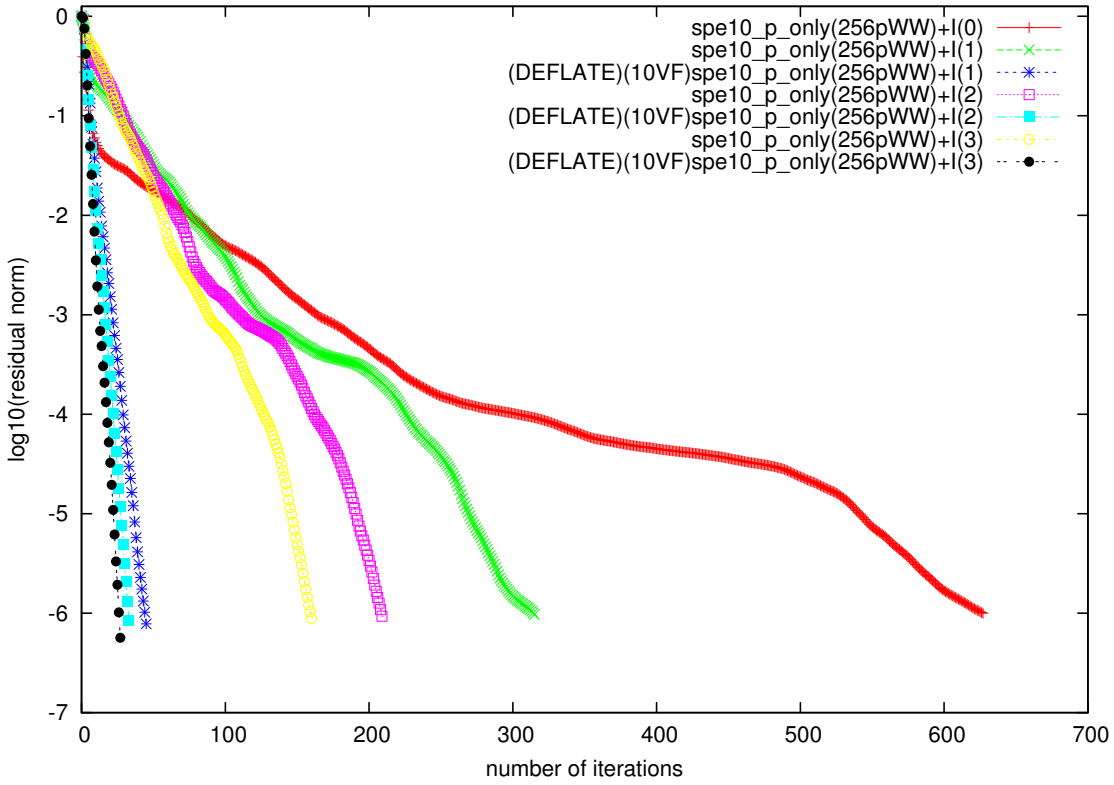
Table 5.23: Results for the matrix: `spe10_p_only.mtx`.

Matrix `spe10_p_only.mtx`

nparts	nrows	nnz
256	1,094,421	7,515,591



(METIS partitioner with weights)



expvar	niter	$\kappa_{\approx}$	nV	$\|r_{sol}\|$	CS[s]	inf[s]	LU[s]	sol[s]
+I(0)	627	54917.50		$9.98e-07$			2.21	60.68
+I(1)	315	7367.61		$2.87e-09$		4.73	5.25	27.33
D..+I(1)	45	28.91	10F	$1.26e-08$	1.05			7.66
+I(2)	209	2529.12		$9.87e-07$		4.87+6.96	11.16	22.19
D..+I(2)	33	16.86	10F	$1.17e-07$	1.64			7.91
+I(3)	160	961.38		$6.85e-10$		4.66+6.97+10.16	20.40	23.80
D..+I(3)	27	13.62	10F	$6.10e-09$	2.27			8.77

Table 5.24: Results for the matrix: `spe10_p_only.mtx` (partitioner with weight).

5.4 IFP Matrix Collection - system of equations

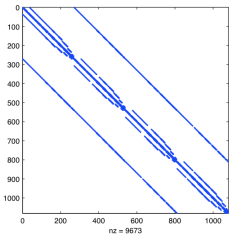
In this subsection we consider full system problem (more then one unknown per cell). We solved each linear system from a given set (see table 5.4) with classical Schwarz method implemented in ADDMlib. We could not use two-level preconditioner since its form (4.6 p.88) was designed for SPD matrixes only. Also partitioning with weights (§2.4) implemented in ADDMlib is limited to scalar case, thus we partitioned matrixes by default $kway$ method from METIS library.

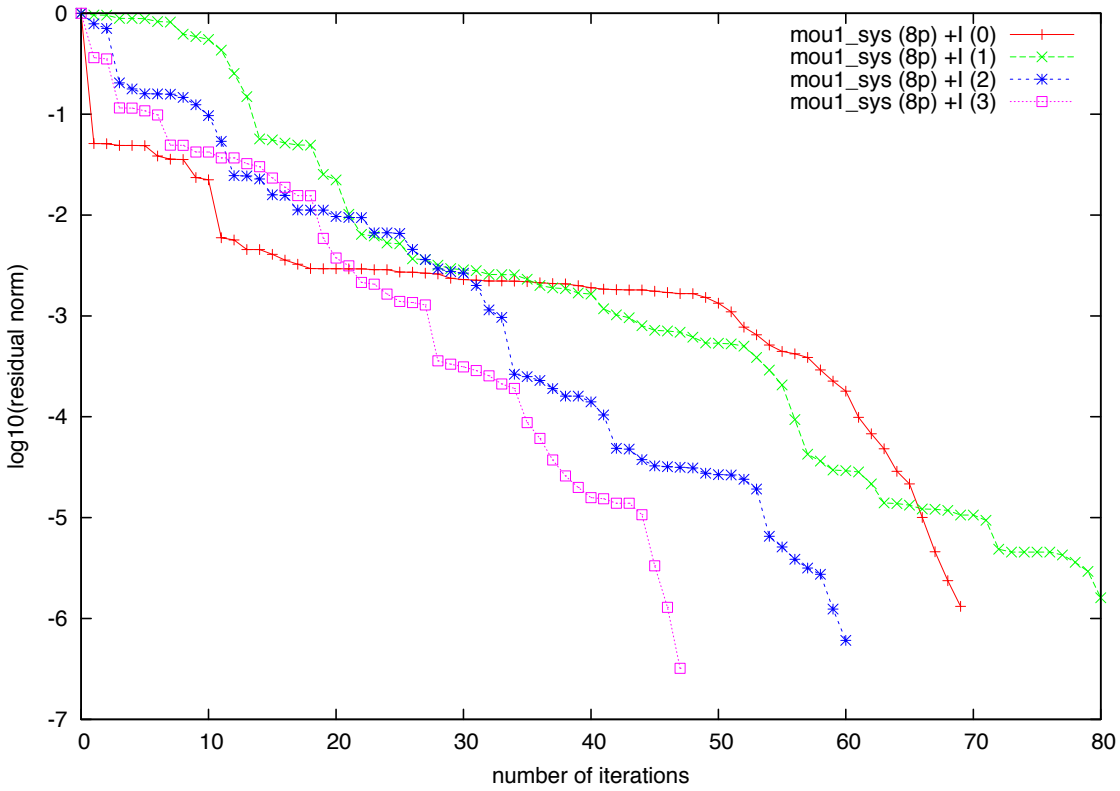
name	**nrows**	**bsize**	**nnz**
mou1	1,080	3	9,673
Canta	24,048	3	146,184
his	104,283	3	1,041,884
IvaskBO	148,716	3	2,417,273
IvaskMULTI	247,860	5	4,461,102
GCS	1,112,946	3	11,544,049
sp10	2,188,842	2	21,554,641

Table 5.25: Matrixes used in experiment, where: **nrow** - number of rows, **bsize** - block size (number of unknowns per mesh cell), **nnz** - number of non-zero elements.

Matrix `mou1_sys.mtx`

nparts	nrows	nnz
8	1,080	9,673



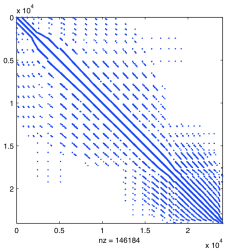


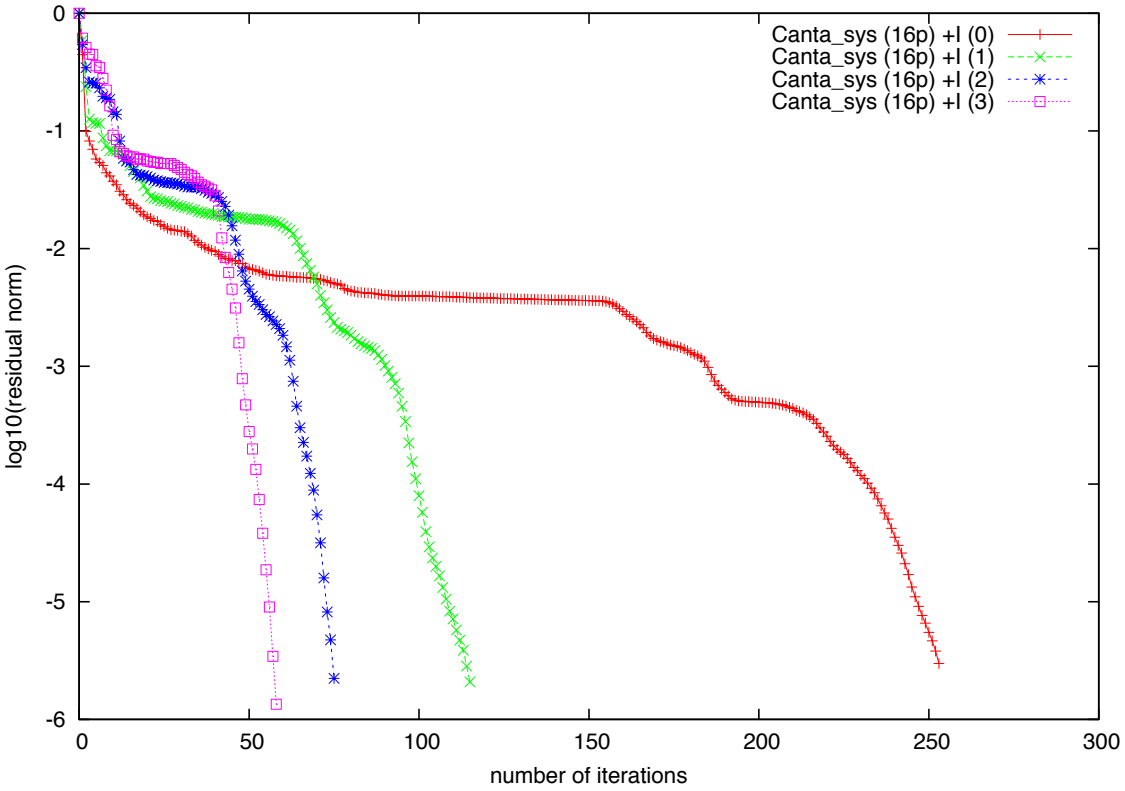
expvar	niter	$\|r_{sol}\|$	inf$[s]$	LU$[s]$	sol$[s]$
+I(0)	70	$4.90e-04$		<1	0.05
+I(1)	81	$2.46e-03$	0.03	<1	0.05
+I(2)	61	$9.81e-03$	0.01+0.02	<1	0.10
+I(3)	48	$1.38e-03$	0.01+0.02+0.02	<1	0.04

Table 5.26: Results for the matrix: `mou1_sys.mtx`.

Matrix `Canta_sys.mtx`

nparts	nrows	nnz
16	24,048	146,184



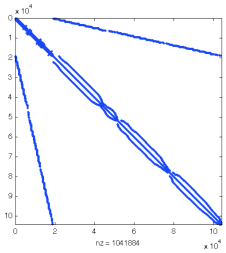


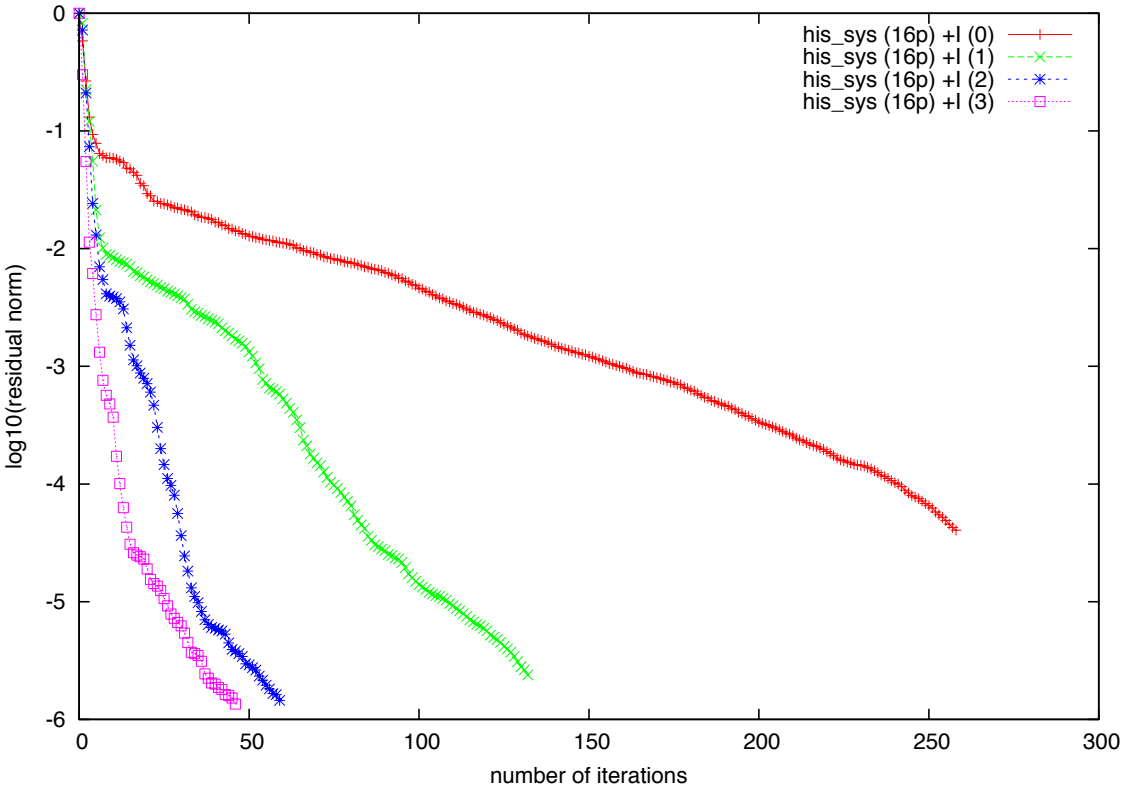
expvar	niter	$\|r_{sol}\|$	inf[s]	LU[s]	sol[s]
+I(0)	254	$3.79e-06$		<1	1.59
+I(1)	116	$8.19e-07$	0.18	<1	0.95
+I(2)	76	$3.32e-07$	0.19+0.31	<1	0.86
+I(3)	59	$3.53e-07$	0.19+0.31+0.5	<1	1.16

Table 5.27: Results for the matrix: `Canta_sys.mtx`.

Matrix `his_sys.mtx`

nparts	nrows	nnz
16	104,283	1,041,884



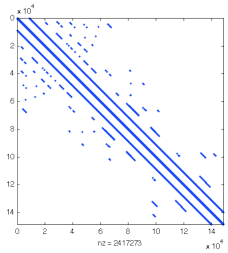


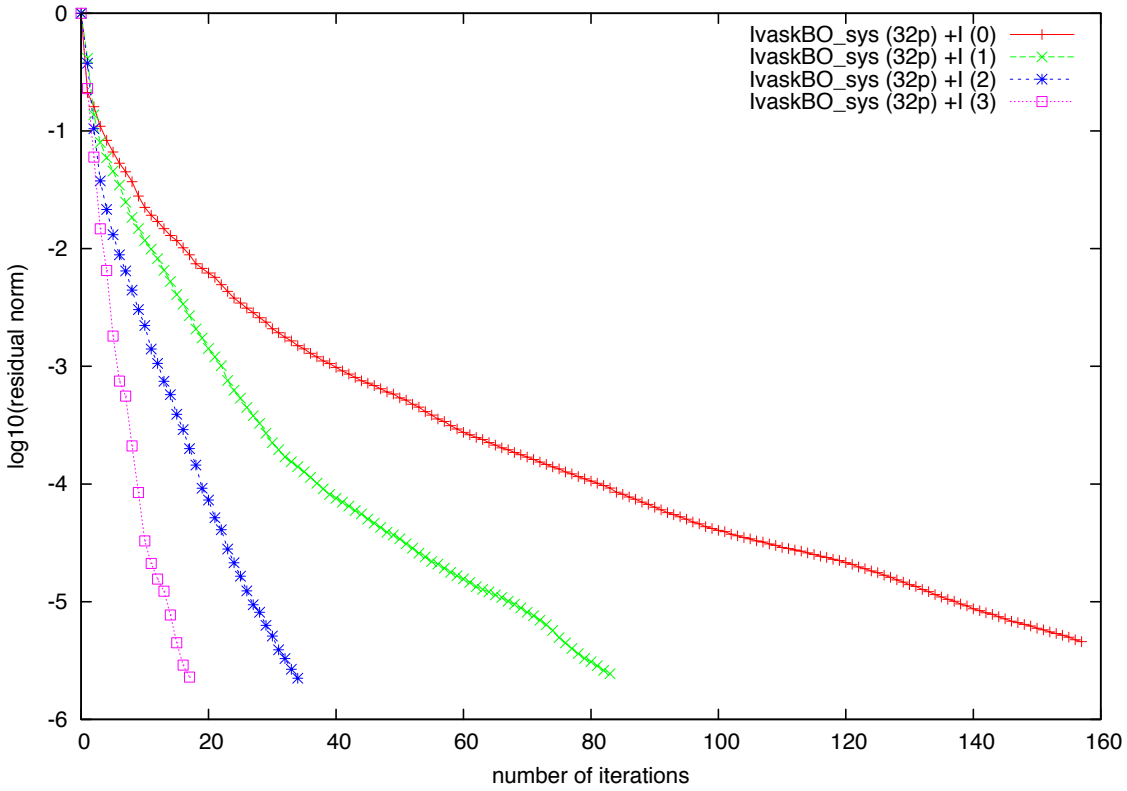
expvar	niter	$\|r_{sol}\|$	inf$[s]$	LU$[s]$	sol$[s]$
+I(0)	259	$1.00e-03$		≈ 1	11.94
+I(1)	133	$1.99e-08$	1.76	≈ 2	6.56
+I(2)	60	$1.08e-08$	1.76+2.18	≈ 2	5.01
+I(3)	47	$2.71e-08$	1.77+2.13+2.89	≈ 4	6.99

Table 5.28: Results for the matrix: `his_sys.mtx`.

Matrix IvaskBO_sys.mtx

nparts	nrows	nnz
32	148,716	2,417,273



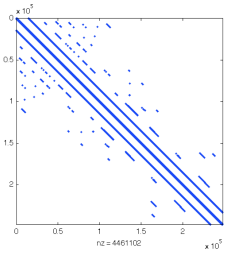


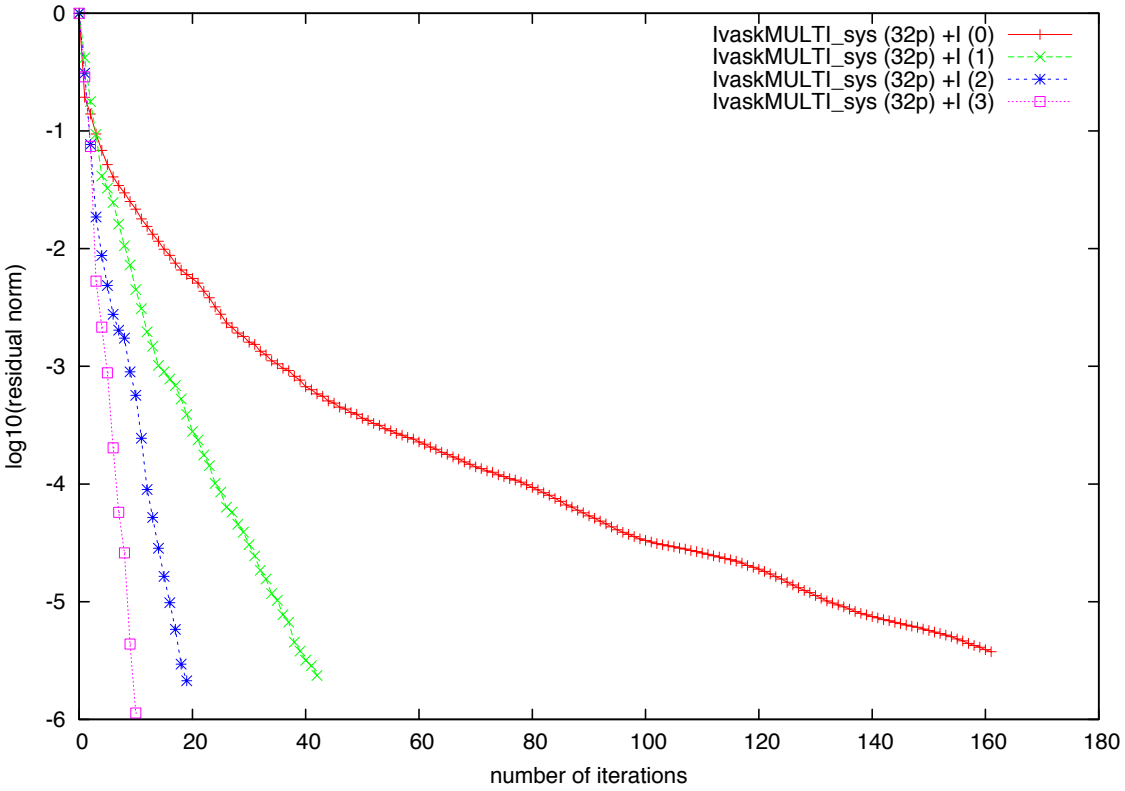
expvar	niter	$\|r_{sol}\|$	inf$[s]$	LU$[s]$	sol$[s]$
+I(0)	158	$2.20e-06$		≈ 1	4.99
+I(1)	84	$4.69e-09$	2.01	≈ 2	6.17
+I(2)	35	$3.58e-09$	2.00+3.29	≈ 4	5.04
+I(3)	18	$2.29e-09$	2.14+3.61+5.35	≈ 8	9.03

Table 5.29: Results for the matrix: IvaskBO_sys.mtx.

Matrix IvaskMULTI_sys.mtx

nparts	nrows	nnz
32	247,860	4,461,102



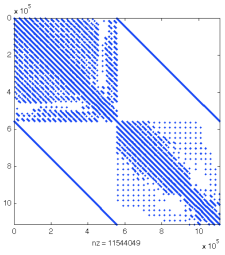


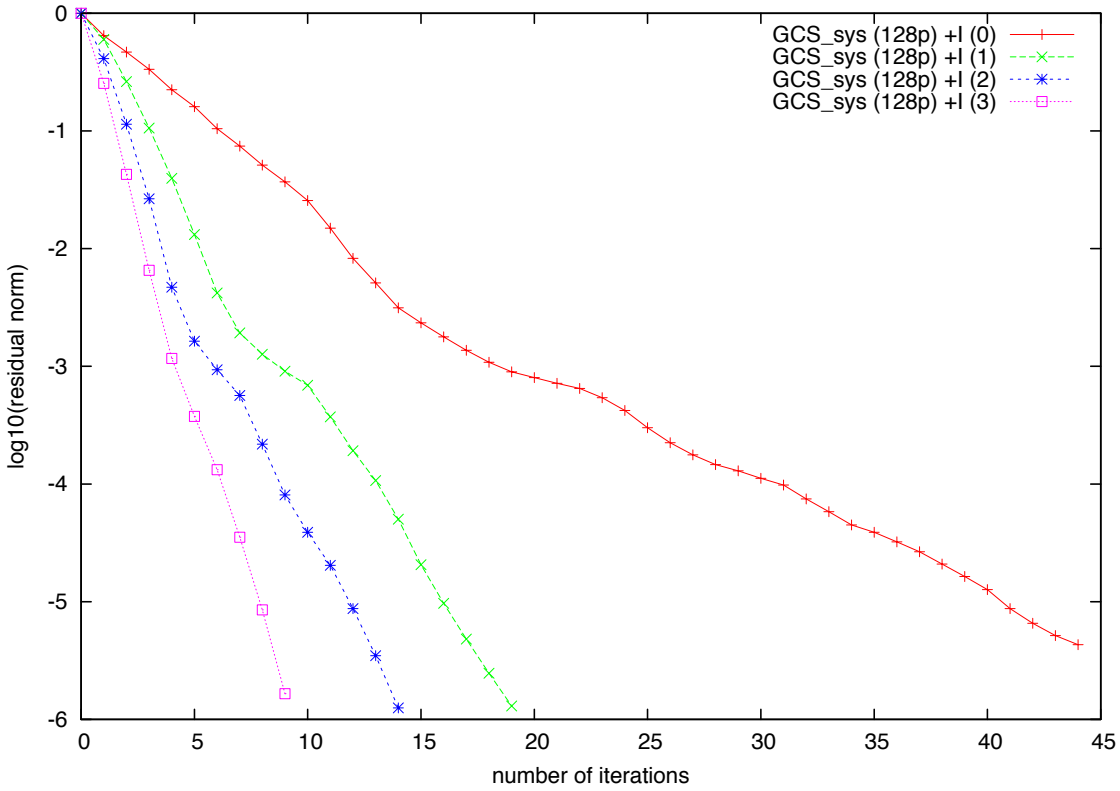
expvar	niter	$\|r_{sol}\|$	inf[s]	LU[s]	sol[s]
+I(0)	162	$1.84e-06$		≈ 5	15.48
+I(1)	43	$2.65e-08$	8.57	≈ 10	14.19
+I(2)	20	$2.90e-08$	8.85+12.84	≈ 21	23.32
+I(3)	11	$8.56e-09$	7.50+10.99+17.40	≈ 35	36.94

Table 5.30: Results for the matrix: IvaskMULTI_sys.mtx.

Matrix GCS_sys.mtx

nparts	nrows	nnz
128	1,112,946	11,544,049



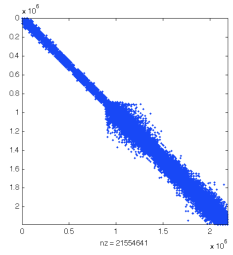


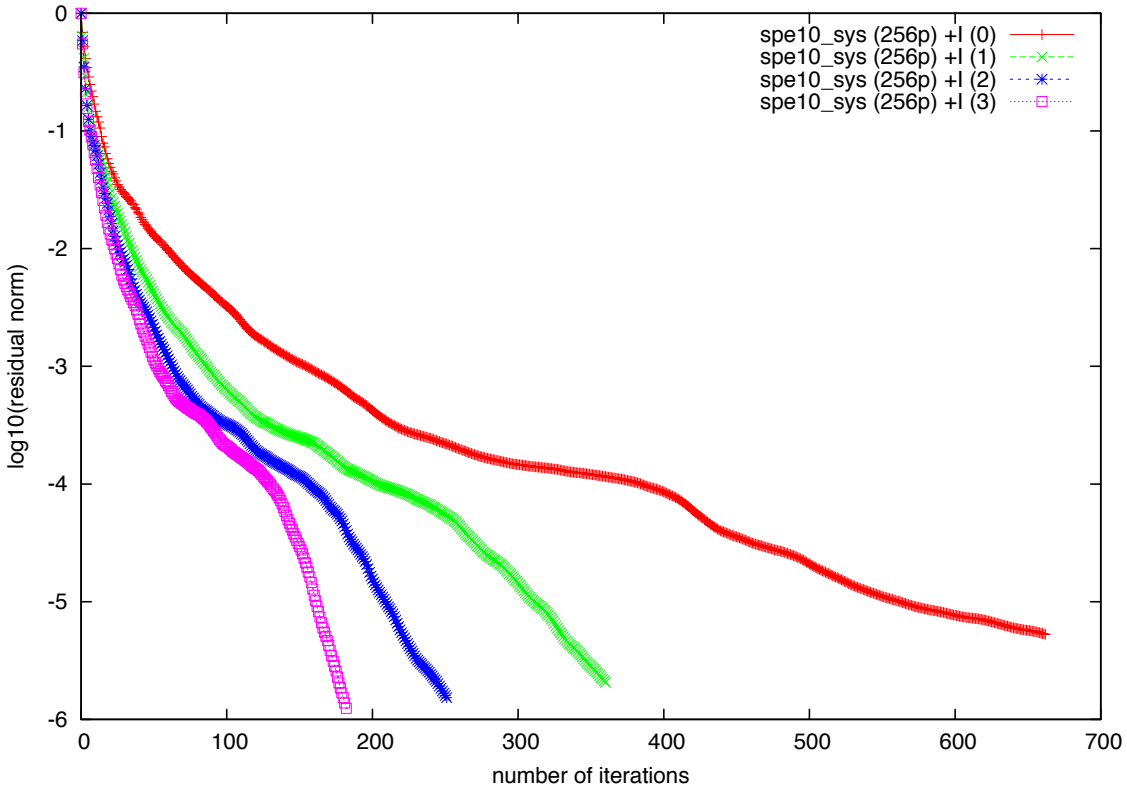
expvar	niter	$\|r_{sol}\|$	inf[s]	LU[s]	sol[s]
+I(0)	45	$3.50e-07$		≈ 9	17.16
+I(1)	20	$1.13e-08$	7.62	≈ 20	24.66
+I(2)	15	$1.58e-08$	7.39+11.82	≈ 47	53.14
+I(3)	10	$2.15e-08$	7.39+11.86+17.66	≈ 78	83.81

Table 5.31: Results for the matrix: GCS_sys.mtx.

Matrix `spe10_sys.mtx`

nparts	nrows	nnz
256	2,188,842	21,554,641





expvar	niter	$\|r_{sol}\|$	inf[s]	LU[s]	sol[s]
+I(0)	663	$1.43e-06$		≈ 4	343.13
+I(1)	361	$2.12e-09$	4.36	≈ 6	70.76
+I(2)	252	$3.81e-08$	4.37 + 7.23	≈ 13	70.88
+I(3)	183	$2.03e-08$	4.52 + 7.42 + 11.00	≈ 24	78.14

Table 5.32: Results for the matrix: `spe10_sys.mtx`.

5.5 Black-Oil Simulation: series of linear systems from Newton algorithm.

The following investigation we dedicate to the solving of large-scale nonlinear problems arising from the finite-volume discretization in which non-linearity is handled by a Newton–Raphson algorithm. We show that we can reuse a coarse operator built from eigenvectors approximated during the first resolution in solutions of linear systems for remaining Newton-Raphson iterations.

Nonlinear problems arising from various applications in mathematics, physics or mechanics. Solving these problems very often leads to a succession of linear problems the solution to which converges towards the solution to the considered problem. In our numerical experiment we consider a series of linear systems extracted from Black-Oil simulator which refer to the Newton-Raphson iterations for a chosen time step in simulation. To be more precise, we consider two series of five linear systems, in which each series correspond to simulation on domain in different size ($60 \times 60 \times 32$ and $120 \times 120 \times 64$ nodes). For each series we performed a following experiment:

- In sequence we read linear system in series from a file and we partition it via graph partitioner (partitioning from first matrix is preserve for future decomposition of remaining matrices in series).
- Resulting `DDMOperator` is once inflate (in order to accelerate convergence).
- Next, we solve inflated linear system. If it is a first linear system in series (`...X_mat1.mtx`) we solve it by Additive Schwarz Method (ASM) and at the end of iterative process we calculate all approximated eigenvectors which corresponding eigenvalues satisfy a following inequality: $Re(\lambda_i) < \text{TOL} = 0.1$. From approximated eigenvectors we construct coarse operator which we preserve for next resolution in series. Hence, if the current linear system was preceded by a coarse space construction we solve it using two-level preconditioner which is a combination of Additive Schwarz preconditioner built for a current linear system and the coarse space operator from first resolution. The schematic view of the stages in our experiment is depicted on figure 5.5.

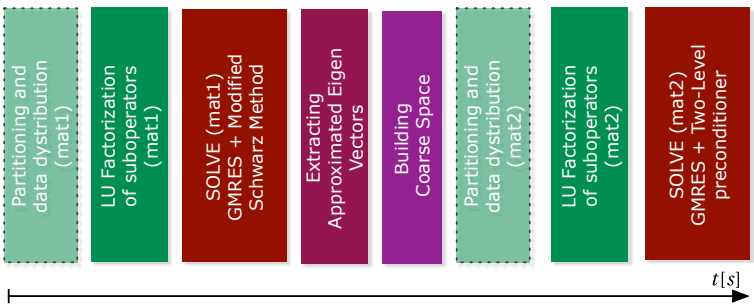


Figure 5.3: Stages of experiment with series of matrices.

All experiment in this section have been performed on UPMC cluster [3] in such a way that two subdomains/parts were dedicated to one MPI process. Therefore in first variant we used 40 processes and in second 80.

5.5.1 Black-Oil - $60 \times 60 \times 32$

Results for series of matrix extracted from Black Oil simulator on regular domain in size $60 \times 60 \times 32$ nodes.

`Matrix BO_60x60x32_matX.mtx`

mat_name	nparts	nrows	nnz
`BO_60x60x32_mat1.mtx`	80	115,200	791,520
`BO_60x60x32_mat2.mtx`	80	115,200	791,572
`BO_60x60x32_mat3.mtx`	80	115,200	791,598
`BO_60x60x32_mat4.mtx`	80	115,200	791,500
`BO_60x60x32_mat5.mtx`	80	115,200	791,512

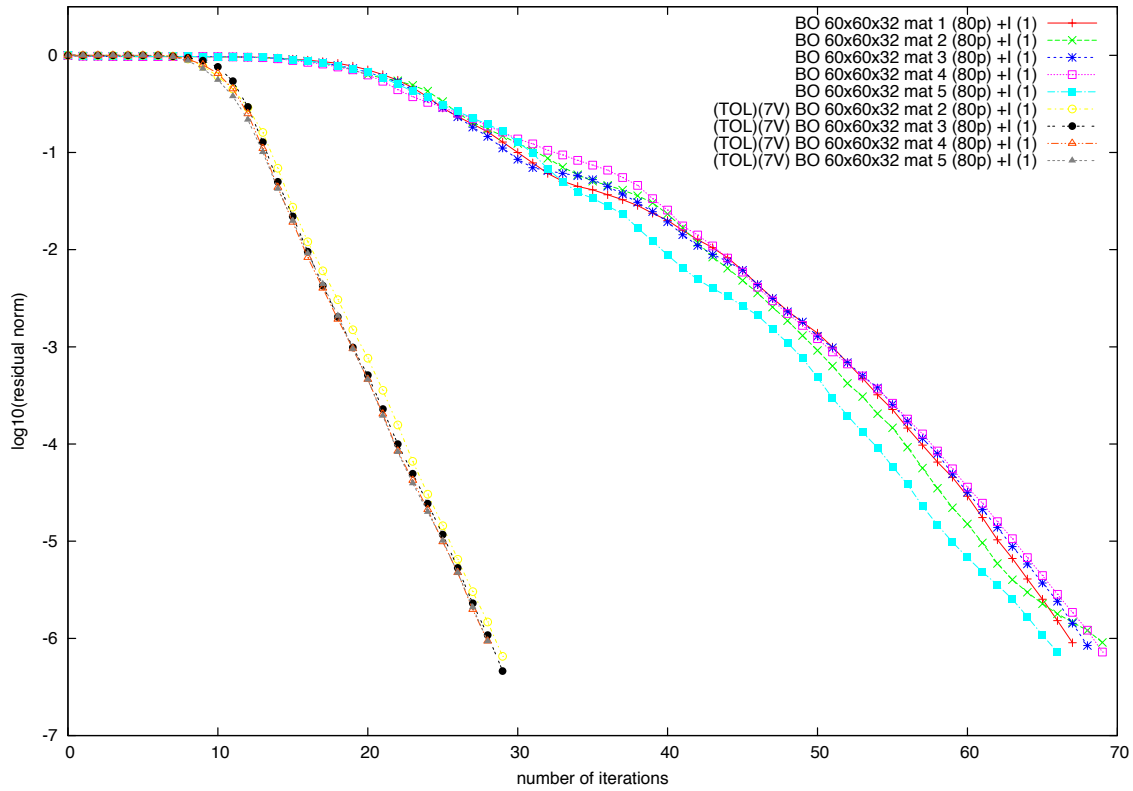


Table 5.33: Convergence curves for series of linear systems solutions with one and two-level preconditioner.

3. See beginning of this chapter for parameters of this super computer.

Prec. type	n-iter
ASM	67
ASM+D (TOL)	28

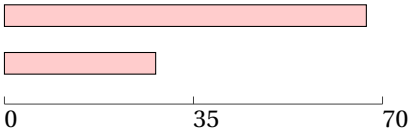


Table 5.34: Comparison of the average number of the iterations for an iterative method with one-level preconditioner (ASM) and two-level preconditioner (ASM+D).

Prec. type		**sol**$[s]$
ASM	$sol_{1\text{lvl}}$	0.42
ASM+D (TOL)	$sol_{2\text{lvl}}$	0.21

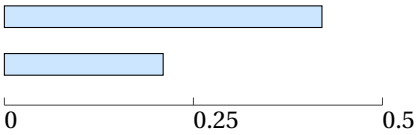


Table 5.35: Comparison of the average time of iterative process for each method (from first to last iteration, without cost of building the preconditioner operator).

	Σ
$5 \times (sol_{1\text{lvl}} + \text{LU}[s])$	2.35
$1 \times (sol_{1\text{lvl}} + \text{LU}[s]) + 4 \times (\text{CS}[s] + \text{LU}[s] + sol_{2\text{lvl}})$	1.59

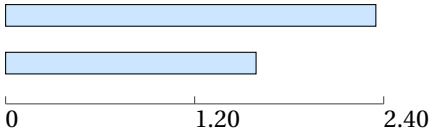


Table 5.36: Comparison of the total time costs of the experiment i.e., time of solving all matrixes with one-level preconditioner versus procedure described at the beginning of this section.

5.5.2 **Black-Oil -** 120 × 120 × 64

Results for series of matrix extracted from Black Oil simulator on regular domain in size 120 × 120 × 64 nodes.

`Matrix BO_120x120x64_matX.mtx`

mat_name	nparts	nrows	nnz
`BO_120x120x64_mat1.mtx`	160	921,600	6,391,680
`BO_120x120x64_mat2.mtx`	160	921,600	6,391,680
`BO_120x120x64_mat3.mtx`	160	921,600	6,391,680
`BO_120x120x64_mat4.mtx`	160	921,600	6,390,986
`BO_120x120x64_mat5.mtx`	160	921,600	6,387,222

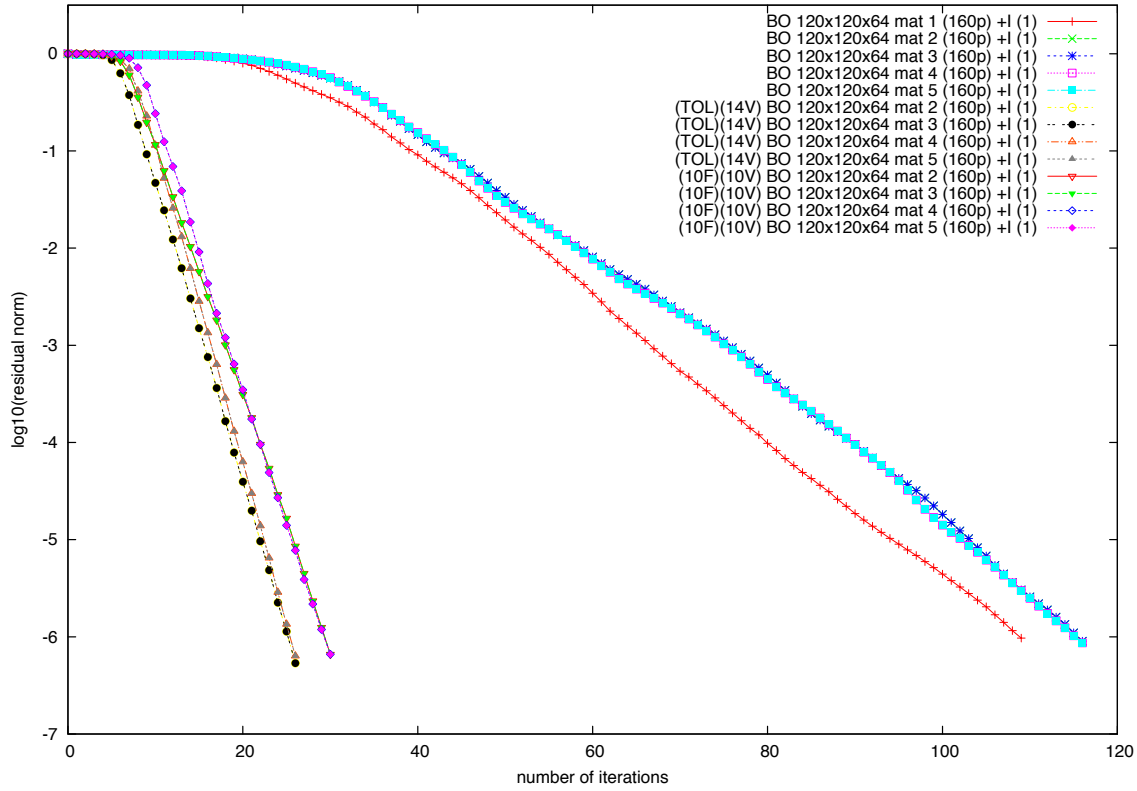


Table 5.37: Convergence curves for series of linear systems solutions with one and two-level preconditioner. Results for series of matrix extracted from Black Oil simulator on regular domain in size 120 × 120 × 64.

Prec. type	n-iter
ASM	114
ASM+D (TOL)	26
ASM+D (10F)	30

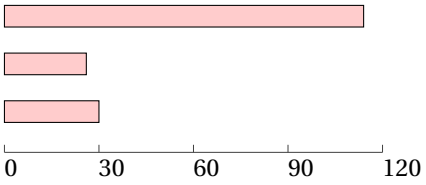


Table 5.38: Comparison of the average number of the iterations for an iterative method with one-level preconditioner (ASM) and two-level preconditioners; (ASM+D TOL) in which for the construction we have used all eigenvectors from the first resolution which corresponding eigenvalues where smaller then TOL = 0.1 and (ASM+D 10V) where only 10 smallest eigenvalues where used.

Prec. type		sol[s]
ASM	sol_{1lvl}	4.18
ASM+D (TOL)	$sol_{\text{1lvl-tol}}$	1.67
ASM+D (10F)	$sol_{\text{1lvl-10F}}$	1.41

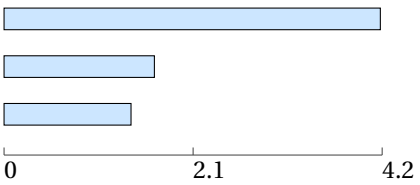


Table 5.39: Comparison of the average time of iterative process for each method (from first to last iteration, without cost of building the preconditioner operator).

	Σ
$5 \times (sol_{\text{1lvl}} + \text{LU}[s])$	26.00
$1 \times (sol_{\text{1lvl}} + \text{LU}[s]) + 4 \times (\text{CS}[s] + \text{LU}[s] + sol_{\text{2lvl-tol}})$	17.92
$1 \times (sol_{\text{1lvl}} + \text{LU}[s]) + 4 \times (\text{CS}[s] + \text{LU}[s] + sol_{\text{2lvl-10F}})$	16.00

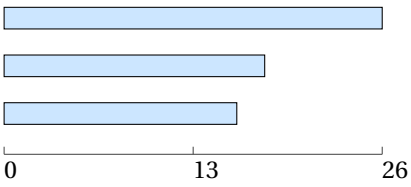


Table 5.40: Comparison of the total time costs of the experiment i.e., time of solving all matrixes with one-level preconditioner versus procedure described at the beginning of this section.

Chapter 6

Conclusion and Prospects

We have focused on algebraic domain decomposition methods in the sense that we have only access to the coefficient of the matrix of the linear system to be solved. We have introduced two new algebraic techniques for building interface conditions and a coarse grid space. Let us mention the important fact that both methods are adaptive and can be used during the first solve that is even before the first solve is completed.

For both methods, we extract information from the Krylov space generated by a few iterations of the Schwarz algorithm with overlapping subdomains. More precisely, we consider the Ritz eigenvectors corresponding to the low eigenvalues since they are responsible for the stagnation or slowness of the Krylov solver preconditioned by the Schwarz algorithm. We are then able to build interface conditions that "kill" the error on the Ritz eigenvector corresponding to the small eigenvalues in magnitude. Numerical tests show that the method brings some benefit but is limited to the two or three subdomain cases. As for the coarse space, it is built by splitting subdomain-wise a given number of the Ritz eigenvectors corresponding to the low eigenvalues. The coarse space is thus larger than the vector space spanned by the Ritz eigenvectors. It contains more information and can be used to complete more efficiently the first solve. Numerical results illustrate the efficiency of the approach even for problems with discontinous coefficients. The tests were made using a libray carefully designed in C++ and MPI with a convenient parallel matrix storage that eases the test of new algorithms. The library uses as much as possible existing libraries (e.g. Metis and Scotch for partitioning, SuperLU, Hypre and PETSC for the sequential linear solvers).

The two new introduced methods have been tested numerically. It was proved that for a symmetric positive definite problem, the algebraic interface conditions lead to submatrices that are still symmetric definite positive. These methods depend on some parameters:

- after how many iterations of the first solve it is best to compute the Ritz eigenvalues,
- how many Ritz eigenvectors should be incorporated in the coarse space.

Numerical tests suggest that the answers to these questions will depend on the difficulty of the problem to be solved. It seems natural as well that for problems arising from systems of partial differential equations, the optimal choices will be different than those for matrices

arising from the pressure block of multiphase porous media flow simulations.

Bibliography

[1] A. Multi-core Strategies: MPI and OpenMP. http://www.hpccommunity.org/f55/multi-core-strategies-mpi-openmpi-702/.

[2] ACHDOU, Y., AND NATAF, F. A robin-robin preconditioner for an advection-diffusion problem. *C. R. Acad. Sci. Paris 325, Série I* (1997), 1211–1216.

[3] ACHDOU, Y., TALLEC, P. L., NATAF, F., AND VIDRASCU, M. A domain decoposition preconditioner for an advection-diffusion problem. *Comp. Meth. Appl. Mech. Engrg 184* (2000), 145–170.

[4] ALDOUS, J., AND WILSON, R. J. *Graphs and Applications: An Introductory Approach*. Springer London Ltd., 2003.

[5] ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, S., DEMMEL, J., DONGARRA, J., DU CROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., AND SORENSEN, D. *LAPACK Users' Guide*, third ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999.

[6] BALAY, S., BUSCHELMAN, K., EIJKHOUT, V., GROPP, W. D., KAUSHIK, D., KNEPLEY, M. G., MCINNES, L. C., SMITH, B. F., AND ZHANG, H. PETSc users manual. Tech. Rep. ANL-95/11 - Revision 3.0.0, Argonne National Laboratory, 2008.

[7] BALAY, S., BUSCHELMAN, K., GROPP, W. D., KAUSHIK, D., KNEPLEY, M. G., MCINNES, L. C., SMITH, B. F., AND ZHANG, H. PETSc Web page, 2009. http://www.mcs.anl.gov/petsc.

[8] BALAY, S., GROPP, W. D., MCINNES, L. C., AND SMITH, B. F. Efficient management of parallelism in object oriented numerical software libraries. In *Modern Software Tools in Scientific Computing* (1997), E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds., Birkhäuser Press, pp. 163–202.

[9] BASTIAN, P., HACKBUSH, W., AND WITTUM, G. Additive and multiplicative multi-grid - a comparison. *Computing 60* (1998), 345–346.

[10] BROQUEDIS, F., CLET ORTEGA, J., MOREAUD, S., FURMENTO, N., GOGLIN, B., MERCIER, G., THIBAULT, S., AND NAMYST, R. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing* (Pisa Italie, 02 2010), IEEE, Ed.

[11] BULUÇ, A., FINEMAN, J. T., FRIGO, M., GILBERT, J. R., AND LEISERSON, C. E. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures* (New York, NY, USA, 2009), ACM, pp. 233–244.

[12] CAI, X.-C., AND SARKIS, M. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM Journal on Scientific Computing 21* (1999), 239–247.

[13] CHAN, T., GLOWINSKI, R., PÉRIAUX, J., AND WIDLUND, O., Eds. *Domain Decomposition Methods* (Philadelphia, PA, 1989), SIAM. Proceedings of the Second International Symposium on Domain Decomposition Methods, Los Angeles, California, January 14 - 16, 1988.

[14] CHAN, T. F., AND MATHEW, T. P. *Acta numerica*. Cambridge University Press, 1994, ch. Domain decomposition algorithms.

[15] CHEN, Z., HUAN, G., AND MA, Y. *Computational Methods for Multiphase Flows in Porous Media*. Society for Industrial and Applied Mathematics, Dallas, Texas, 2006.

[16] CHEVALIER, C., AND PELLEGRINI, F. PT-SCOTCH: a tool for efficient parallel graph ordering. *Parallel Computing 6-8*, 34 (2008), 318–331.

[17] COLLINO, F., GHANEMI, S., AND JOLY, P. Domain decomposition methods for harmonic wave propagation : a general presentation. *Comput. Methods Appl. Mech. Engrg*, 2-4 (2000), 171–211.

[18] DEMMEL, J. W., EISENSTAT, S. C., GILBERT, J. R., LI, X. S., AND LIU, J. W. H. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Application 20*, 3 (1999), 720–755.

[19] DESPRÉS, B. Domain decomposition method and the Helmholtz problem.II. In *Second International Conference on Mathematical and Numerical Aspects of Wave Propagation (Newark, DE, 1993)* (Philadelphia, PA, 1993), SIAM, pp. 197–206.

[20] DRYJA, M., AND WIDLUND, O. B. Towards a unified theory of domain decomposition algorithms for elliptic problems. In *Third International Symposium on Domain Decomposition Methods for Partial Differential Equations* (1990), T. Chan, R. Glowinski, J. Périaux, and O. B. Widlund, Eds., SIAM, Philadelphia, PA, pp. 3–21.

[21] DURLOFSKY, L. J. A triangle based mixed finite element-finite volume technique for modeling two phase low through porous media. *J. Comput. Phys.*, 105 (1993), 252–266.

[22] EFSTATHIOU, E., AND GANDER, M. J. Why Restricted Additive Schwarz converges faster than Additive Schwarz. *BIT Numerical Mathematics 43* (2003), 945–959.

[23] ERLANGGA, Y. A., AND NABBEN, R. Deflation and balancing preconditioners for krylov subspace methods applied to nonsymmetric matrices. *SIAM J. Matrix Anal. Appl. 30* (2008), 684–699.

[24] FLAURAUD, E., NATAF, F., AND WILLIEN, F. Optimized interface conditions for domain decomposition methods for problems with extreme contrast in the coefficients. *Journal of Computational and Applied Mathematics 189* (2006), 539–554.

[25] GAMMA, E. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2004.

[26] GANDER, M. J. Schwarz methods in the course of time. *ETNA 31* (2008), 228–255.

[27] GANDER, M. J., HALPERN, L., MAGOULÈS, F., AND ROUX, F. X. Analysis of patch substructuring methods. *Int. J. Appl. Math. Comput. Sci. 17*, 2 (2007), 395–402.

[28] GANDER, M. J., MAGOULÈS, F., AND NATAF, F. Optimized Schwarz methods without overlap for the Helmholtz equation. *SIAM J. Sci. Comput. 24*, 1 (2002), 38–60.

[29] GOOSSENS, S., AND ROOSE, D. Ritz and harmonic ritz values and the convergence of fom and gmres. *Numerical Linear Algebra with Applications 6*, 4 (Sep 1999), 281–293.

[30] GOSSELET, P., AND REY, C. On a selective reuse of Krylov subspaces in Newton-Krylov approaches for nonlinear elasticity. In *Proceedings of the fourteenth international conference on domain decomposition methods Fourteenth international conference on domain decomposition methods* (Cocoyoc Mexique, 2003), O. W. I. Herrera, D. Keyes and R. Yates, Eds., pp. 419–426.

[31] GREENBAUM, A. *Iterative Methods for Solving Linear Systems*. SIAM, 1997.

[32] GROP, W., LUSK, E., AND SKJELLUM, A. *Using MPI: Portable Parallel Programming with Message-Passing Interface*. MIT Press, 1994.

[33] HAGSTROM, T., TEWARSON, R. P., AND JAZCILEVICH, A. Numerical experiments on a domain decomposition algorithm for nonlinear elliptic boundary value problems. *Appl. Math. Lett. 1*, 3 (1988).

[34] HECHT, F. *FreeFem++*, 3.7 ed. Numerical Mathematics and Scientific Computation. Laboratoire J.L. Lions, Université Pierre et Marie Curie, http://www.freefem.org/ff++/, 2010.

[35] HESTENES, M., AND STIEFEL, E. Methods of conjugate gradient for solving linear systems. *J. Res. Nat. Bur. Stand. 49* (1952), 409–436.

[36] KARYPIS, G., AND KUMAR, V. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing 20*, 1 (1999), 359–392.

[37] LAI, C.-H., BJØRSTAD, P. E., CROSS, M., AND WIDLUND, O., Eds. *Eleventh International Conference on Domain Decomposition Methods* (1998). Proceedings of the 11th International Conference on Domain Decomposition Methods in Greenwich, England, July 20-24, 1998.

[38] LE TALLEC, P. Domain decomposition methods in computational mechanics. In *Computational Mechanics Advances*, J. T. Oden, Ed., vol. 1 (2). North-Holland, 1994, pp. 121–220.

[39] LIONS, P.-L. On the Schwarz alternating method. III: a variant for nonoverlapping subdomains. In *Third International Symposium on Domain Decomposition Methods for Partial Differential Equations ,held in Houston, Texas, March 20-22, 1989* (Philadelphia, PA, 1990), T. F. Chan, R. Glowinski, J. Périaux, and O. Widlund, Eds., SIAM.

[40] MAGOULÈS, F., ROUX, F.-X., AND SALMON, S. Optimal discrete transmission condition for non-overlapping domain decomposition method for helmholtz equation. *SIAM Journal on Scientific Computing 25*, 5 (2004), 1947–1515.

[41] MAGOULÈS, F., ROUX, F. X., AND SERIES, L. Algebraic approximation of dirichlet-to-neumann maps for equations of linear elasticity. *Comp. Meth. Appl. Mech. Engrg. 195* (2006), 3742–3759.

[42] MANDEL, J. Balancing domain decomposition. *Communications in Applied and Numerical Methods 9* (1993), 233–241.

[43] MATSOKIN, A. M., AND NEPOMNYASCHIKH, S. V. A Schwarz alternating method in a subspace. *Soviet Mathematics 29(10)* (1985), 78–84.

[44] NATAF, F., AND ROGIER, F. Factorization of the convection-diffusion operator and the Schwarz algorithm. M^3AS *5*, 1 (1995), 67–93.

[45] NATAF, F., ROGIER, F., AND DE STURLER, E. Optimal interface conditions for domain decomposition methods. Tech. Rep. 301, CMAP (Ecole Polytechnique), 1994.

[46] NATAF, F., XIANG, H., AND DOLEAN, V. A coarse space construction based on local dtn maps. http://hal.archives-ouvertes.fr/hal-00491919/fr/, 2010.

[47] NICOLAIDES, R. A. Deflation of conjugate gradients with applications to boundary value problems. *SIAM J. Matrix Anal. Appl. 24* (1987), 355–365.

[48] PADIY, A., AXELSSON, O., AND POLMAN, B. Generalized augmented matrix preconditioning approach and its application to iterative solution of ill-conditioned algebraic systems. *SIAM J. Matrix Anal. Appl. 22* (2000), 793–818.

[49] PETTER E. BJØRSTAD, M. S. E., AND KEYES, D. E., Eds. *Ninth International Conference on Domain Decomposition Methods* (1997). Proceedings of the 9th International Conference on Domain Decomposition Methods in Bergen, Norway.

[50] QUARTERONI, A., AND VALLI, A. *Domain Decomposition Methods for Partial Differential Equations*. Oxford Sci. Publ., Oxford University Press, 1999.

[51] SAAD, Y. *Iterative Methods for Sparse Linear Systems*, 2nd ed. SIAM, Philadelphia, 2003.

[52] SAAD, Y., AND SCHULTZ, M. H. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput. 7*, 3 (1986).

[53] SCHWARZ, H. A. Über einen Grenzübergang durch alternierendes Verfahren. *Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich 15* (May 1870), 272–286.

[54] SMITH, B. F., BJØRSTAD, P. E., AND GROPP, W. *Domain decomposition: Parallel Multilevel Methods for Eliptic Partial Differential Equations*. Cambridge University Press, 1996.

[55] STROUSTRUP, B. *The* C++ *Programming Language*. Addison-Wesley, 1997.

[56] STÜBEN, K. A review of algebraic multigrid. *Journal of Computational and Applied Mathematics 128* (2001), 281–309.

[57] TANG, J. M., NABBEN, R., VUIK, C., AND ERLANGGA, Y. A. Comparison of two-level preconditioners derived from deflation, domain decomposition and multigrid methods. *J. Sci. Comput. 39* (2009), 340–370.

[58] TOSELLI, A., AND WIDLUND, O. *Domain Decomposition Methods: algorithms and theory*. Springer, 2005.